TAMPEREEN TEKNILLINEN YLIOPISTO

MIIKA JÄRVINEN

PLAGIOINNIN AUTOMAATTINEN ETSINTÄ LÄHDEKOODISTA

Diplomityö

Tarkastaja: Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkö tiedekuntaneuvoston
kokouksessa 03.10.2012

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
MIIKA JÄRVINEN: Plagioinnin automaattinen etsintä lähdekoodista
Diplomityö, 44 sivua
Helmikuu 2014
Pääaine: Ohjelmistotuotanto
Tarkastajat: Tommi Mikkonen
Avainsanat: Plagiointi, lähdekoodi

Plagioinnin järjestelmällinen tunnistaminen suuresta määrästä lähdekooditiedostoja on vaikeaa. Se vaatii kaikkien tarkasteltavana olevien lähdekooditiedostojen vertaamista toisiinsa, josta muodostuu jo pienillä tiedostomäärillä suuri työ. Lisäksi plagioitua materiaalia on voitu muokata työn alkuperän piilottamiseksi, jolloin plagioinnin havaitseminen vaatii vertailtavien töiden samankaltaisuuden tulkintaa. Tunnistamista voidaan helpottaa seulomalla lähdekooditiedostoista toisiaan muistuttavia pareja samankaltaisuutta mittaavalla tietokoneohjelmalla.

Tässä diplomityössä määritellään plagioinnin tunnistamisen vaatimukset sekä kuvataan koulutusorganisaatioille sopiva plagioinnin tunnistuksen prosessi, jota noudattamalla voidaan plagiointeja etsiä ja plagiointiepäilyjä käsitellä järjestelmällisesti. Työssä esitellään tekniset ratkaisut, joiden avulla tunnistaminen voidaan tehdä, vaikka plagioidun materiaalin alkuperää olisi peitelty. Lisäksi työssä on toteutettu plagioinnin automaattiseen etsintään soveltuva työkalu Plakki 2.1, jonka suoriutumista on arvioitu vertaamalla sen tuloksia kahden muun vastaavan työkalun antamiin tuloksiin.

Plagioinnin tunnistaminen luotettavasti suuresta määrästä lähdekooditiedostoja vaatii tuekseen järjestelmällisen toimintaprosessin, jonka apuna voidaan käyttää lähdekoodin samankaltaisuutta arvioivaa tietokoneohjelmaa. Plagiointia ei voida todeta pelkästään automaattisen etsinnän tulosten perusteella, vaan etsinnän suorittajan täytyy tuntea plagioinnin naamiointiin käytetyt menetelmät ja huolellisesti varmistaa plagiointiepäily etsimällä plagiointiin viittaavia todisteita. Työssä esitetyt tekniset ratkaisut ja niitä soveltava työkalu ovat suorituskyvyltään ja tarkkuudeltaan kilpailukykyisiä muihin vastaaviin järjestelmiin verrattuna.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
MIIKA JÄRVINEN: Automatic Search of Plagiarism in Source Code
Master of Science Thesis, 44 pages
February 2014
Major: Software Engineering
Examiner: Tommi Mikkonen
Keywords: Plagiarism, source code

The systematic identification of plagiarism from an extensive amount of source code is difficult. It requires that every source code file is compared to all other source code files included in comparison. This results in a considerable amount of work even with a small number of files. In addition, plagiarized material may have been modified to disguise the true origin of source code. Consequently, interpretation of similarity is necessary. The identification of plagiarism can be facilitated by computer programs designed to trace similarities in source code.

This thesis defines plagiarism and the process of plagiarism verification. This process is exemplified in the context of an educational institution. This thesis aims to provide a guideline of how to search for and handle cases of possible plagiarism in a systematic way. This thesis presents technical solutions of how to identify plagiarism also in cases where plagiarism has been disguised. This thesis presents a redesigned computer software tool Plakki 2.1, which can be used in search for plagiarism automatically. The performance of this tool has been evaluated by comparing it with two similar tools.

The identification of plagiarism reliably from an extensive amount of source code requires a systematic approach. This includes a computer program designed to search for similarities in source code. Nonetheless, the identification of similarities as plagiarism always requires manual interpretation. It requires knowledge of methods used to disguise plagiarism and careful inspection of evidence indicating plagiarism. The technical solutions and the redesigned tool Plakki 2.1 are compatible in efficiency and precision with other similar systems.

ALKUSANAT

Tässä diplomityössä käsitellään menetelmiä ja tekniikoita, joilla voidaan automaattisesti etsiä plagiointia lähdekoodista. Diplomityössä esitellään myös työkalu nimeltä Plakki 2.1, jolla etsintää voidaan tehdä. Plakkia 2.1 on työssä verrattu kahteen muualla toteutettuun plagiaatinpaljastimeen. Plakki 2.1 on uudistettu versio vanhasta Tampereen Teknillisen Yliopiston plagiaatinpaljastimesta, jonka alkuperäisinä tekijöinä ovat Ari Suntioinen, Toni Uimonen ja Kirsti Ala-Mutka.

Haluan kiittää Tommi Mikkosta diplomityön lukemisesta, ohjauksesta ja työn aiheen ehdottamisesta. Lisäksi kiitän Samuel Lahtista, Juho Lauria ja Joona Järvistä, jotka pyynnöstäni toimittivat käyttööni plagioituja versioita kirjoittamistani ohjelmista.

Tampereella 17.1.2014

Miika Järvinen

SISÄLLYS

MATEMAATTISET SYMBOLIT

Symbolit

$O(f(x))$	O -notaatio. Kuvaa jonkin funktion kasvunopeutta syötteen suhteen antamalla ylärajan kasvunopeudelle yksinkertaisemman funktion $f(x)$ avulla.
$\langle ... \rangle$	Rajaa yhden alkion muodostumista, jos alkio koostuu useammasta arvosta.
$\cap$	Joukkojen leikkaus. Joukkoon $A \cap B$ kuuluu joukkojen A ja B yhteiset alkiot.
$\{x \mid \varphi(x)\}$	Joukon muodostuminen. Joukkoon kuuluu sellaiset alkiot x, joille pätee ehto $\varphi(x)$. Joukon muodostuminen voidaan käyttää myös muodossa { ... }, jolloin joukkoon kuuluvat kaikki aaltosulkeiden sisällä luetellut alkiot.[12]
$\forall$	Kaikki-kvanttori. $\forall x : \varphi(x)$ tarkoittaa, että kaikille alkioille x pätee ehto $\varphi(x)$. [12]
$\in$	Joukkoon kuuluminen.
$\wedge$	Looginen ja-operaattori.
x_i	Tarkoittaa muuttujaa, joka on yksilöity alaindeksillä i.
X_i	Tarkoittaa jonoa, jossa on i alkiota.
$<$	Pienempi kuin. $x < y$ tarkoittaa, että x on pienempi kuin y.
$\leq$	Pienempi tai yhtä suuri kuin.
Σ	Summa. $\sum_{\varphi(x)} f(x)$ tarkoittaa, että kaikille alkioille x, jotka toteuttavat ehdon $\varphi(x)$, suoritetaan lauseke $f(x)$ ja tulokset lasketaan yhteen.
$min(x, y)$	Tarkoittaa pienempää tai yhtä suurta kahdesta arvoista x ja y, joille on määritelty operaattori $<$.
$=$	Yhtäsuuruus, samuus.
$\neq$	Erisuuruus, ei sama.
$\vee$	Looginen tai-operaattori.

$\setminus$	Joukkojen miinus-operaatio. $A \setminus B$ tarkoittaa sellaista joukkoa, jossa on sellaiset joukon A alkiot, joita ei ole joukossa B.
$\leftarrow$	Sijoitus. $x \leftarrow y$ tarkoittaa, että muuttuja x saa arvon y.
.	Tietueen tai olion kentän valitsin.
$lenght(x)$	Muuttujan x pituus.
$Array[1..n]$	n-alkioisen taulukon luonti. Taulukon arvot ovat oletusarvoisesti määrittelemättömiä.
$A[i]$	Taulukon indeksointi. Tarkoittaa taulukon A alkiota kohdassa i.
$max(x, y)$	Tarkoittaa suurempaa tai yhtäsuurta arvoista x ja y, joille on määritelty operaattori $<$.
$swap(x, y)$	Vaihtaa muuttujat x ja y toisinpäin. $swap(A, B)$, jossa A ja B ovat taulukoita, tarkoittaa viittauksen vaihtoa, eli taulukoiden sisältöä ei kopioida.
$compare(x, y)$	Tarkoittaa, että olioille x ja y suoritetaan niiden välinen vertailu erilaisilla vertailualgoritmeilla.

1. JOHDANTO

Tietokoneohjelmia kirjoitetaan erilaisilla ohjelmointikielillä, joiden kielioppi ja semantiikka on tarkasti määritelty. Tietokone ei kykene suorittamaan ohjelmaa ohjelmointikielellä kirjoitetusta *lähdekoodista* suoraan, vaan lähdekoodi on ennen suorittamista käännettävä tietokoneen ymmärtämiksi käskyiksi siihen erityisesti tarkoitetulla ohjelmalla: kääntäjällä tai tulkilla. Tietokoneohjelman käyttäytyminen eli toiminnallisuus seuraa ohjelman lähdekoodista, mutta sama toiminnallisuus voidaan yleensä saavuttaa monella erilaisella tavalla kirjoitetulla lähdekoodilla.

Ohjelmien lähdekoodit sisältävät samankaltaisuuksia silloin, kun yhdestä lähdekoodista on tarkoituksellisesti kirjoitettu tai kopioitu useampia versioita, mutta samankaltaisuuksia voi ilmetä myös sattumalta. Ohjelmien lähdekoodeista pyritään yleensä löytämään sellaisia samankaltaisuuksia, jotka eivät ole oikeutettuja, eli lähdekoodi tai sen osa on *plagioitu*.

Tämä diplomityö käsittelee menetelmiä, joilla ohjelmien lähdekoodeista voidaan etsiä plagioituja osia. Työssä esitetään tekniset ratkaisut, joilla plagiointia voidaan tunnistaa, vaikka alkuperäistä lähdekoodia olisi muokattu siten, ettei sen alkuperä olisi jäljitettävissä. Työn tavoitteena on kuvata koulutusorganisaatioiden käyttöön soveltuva plagioinnin etsinnän prosessi hallinnollisella ja teknisellä tasolla sekä tuottaa työkalu, jolla samankaltaisuuksia voidaan etsiä lähdekooditiedostoista automaattisesti. Tuotetun työkalun suoriutumista arvioidaan vertaamalla sen antamia tuloksia kahden vastaavanlaisen työkalun tuloksiin erilaisilla testijoukoilla, jotka on muodostettu Tampereen teknillisen yliopiston perusohjelmoinnin kurssien harjoitustyöpalautuksista. Testijoukkoihin on lisätty keinotekoisia plagiaatteja TTY:n henkilökunnan toimesta, jotta todellista plagiaattien tunnistuskykyä voidaan arvioida.

Työn aihe syntyi keväällä 2011 Googlen ja Oraclen välisestä tekijänoikeuskiistasta, jolloin professori Tommi Mikkonen ehdotti Googlen Androidin ja Oraclen Java SE:n vertailemista TTY:n ohjelmistotekniikan laitoksella toteutetulla automaattisen plagioinnin etsinnän työkalulla nimeltä Plakki 1.0 [11]. Vertailua suoritettaessa huomattiin Plakin 1.0 tehottomuus suurten tietomäärien seulonnassa, jolloin Plakista 1.0 päätettiin tehdä uusi versio, jossa tehokkuusongelmat olisi ratkaistu. Samalla

työkalun toimintaa ja suoriutumista vertailun paikkansapitävyyden osalta voitaisiin tarkistaa. Plakista 1.0 ei oltu tuotettu juurikaan teknistä dokumentaatiota, joten järjestelmän toiminta täytyi tunnistaa pelkästään ohjelmakoodin ja järjestelmän toiminnan perusteella. Plakista 1.0 toteutettiin uudistettu versio 2.1 TTY:n ohjelmistotekniikan laitoksen käyttöön. Plakin 2.1. toteuttamisen jälkeen havaittiin tarve plagioinnin etsinnän prosessin kuvaamiselle, jolloin etsinnän käytäntöjä voitaisiin yhtenäistää TTY:n kurssien osalta.

Luvussa 2 esitellään ohjelmistojen samankaltaisuuksia yleisesti ja käsitellään erityisesti plagiointia ja syitä plagiointiin. Olennainen osa plagioinnin tunnistusta on ymmärtää plagioijan motiivit, jolloin mahdollinen plagioinnin yritys voidaan vielä estää. Luvussa 3 esitellään vaatimukset plagioinnin tunnistamiselle ja prosessi, jota suorittamalla plagiointitapauksia voidaan etsiä järjestelmällisesti. Luvussa 4 käsitellään teoreettiselta näkökannalta vaatimukset täyttäviä ratkaisuvaihtoehtoja ja luvussa 5 kuvataan teknisestä näkökulmasta plagiaattien etsintään tarkoitettu työkalu Plakki 2.1. Luvussa 6 esitellään muut järjestelmät, jotka valittiin automaattisen plagioinnin tunnistuksen työkalujen vertailuun, määritellään vertailun tavoitteet ja kuvataan käytetyt testijoukot. Lopuksi käsitellään vertailun tulokset. Luvussa 7 käsitellään yhteenveto.

2. OHJELMISTOJEN SAMANKALTAISUUDET

Ohjelmistoilla on usein paljon rakenteellisia samankaltaisuuksia, jotka voivat olla laillisesta näkökulmasta joko hyväksyttäviä tai kiellettyjä. Ohjelmistonkehityksessä pyritään yleensä yleiskäyttöisiin ja uudelleenkäytettäviin ratkaisuihin, jotka ovat luonnollisesti hyväksyttäviä samankaltaisuuksia. Hyväksyttäviä samankaltaisuuksia ovat myös tekijän- ja omistusoikeudellisista syistä uudelleen kirjoitetut, mutta saman toiminnallisuuden sisältävät ohjelmakomponentit. Kielletyt samankaltaisuudet ovat sellaisia samankaltaisuuksia, jotka on tuotettu jonkun toisen omistamasta lähdekoodista ilman asianmukaista korvausta tai sopimusta.

Tässä luvussa esitellään erilaisista lähtökohdista seuranneita samankaltaisuuksia ja käsitellään yksityiskohtaisemmin koulutusorganisaatioissa ilmeneviä samankaltaisuuksia.

2.1 Hyväksyttävät samankaltaisuudet

Tässä kohdassa esitellään lyhyesti hyväksyttävät samankaltaisuudet, koska samankaltaisuuksien etsijän täytyy pystyä erottamaan hyväksyttävät ja kielletyt samankaltaisuudet toisistaan.

Monistunut ohjelmakoodi. Monistunut ohjelmakoodi tarkoittaa sitä, että sama toiminnallisuus on useassa eri paikassa, joten toiminnallisuutta päivitettäessä päivitykset täytyy tehdä useampaan paikkaan. Monistunut ohjelmakoodi ei sinällään ole laitonta oikeuksien puolesta, mutta sitä pidetään yleensä haitallisena ohjelmiston ylläpidon kannalta.

Ohjelmistojen yhteiset osat. Ohjelmistojen yhteiset osat ovat monistuneen ohjelmakoodin erityistapaus useamman ohjelman kesken. Seuraukset ohjelmistojen yhteisissä osissa ovat kuitenkin erilaiset. Yhteiset osat voivat olla esimerkiksi kolmannen osapuolen tuottamia palveluita, kuten käytetyn ohjelmointikielen kirjastot. Ohjelmistojen yhteisten osien tunnistaminen voi olla tarpeen niitä päivitettäessä ja riippuvuuksien määrittelyssä.

Sama tavoite. Ohjelmistoilla saattaa olla tavoitteena saman toiminnallisuuden toteuttaminen, jolloin samankaltaisuudet ovat jo abstraktilla vaatimustasolla. Samat toiminnallisuusvaatimukset eivät takaa rakenteellisia samankaltaisuuksia, mutta voivat johtaa sellaisiin. Ohjelmointikurssien harjoitustyöt ovat esimerkkitapauksia ohjelmistoista, joilla on samat toiminnalliset vaatimukset. Samoja tavoitteita on myös ohjelmistoteollisuudessa, kun kaksi yritystä tuottaa keskenään kilpailevaa tuotetta tai hyödykettä.

2.2 Plagiointi

Termillä plagiointi tarkoitetaan yhteisesti kaikkia kiellettyjä samankaltaisuuksia. Suomen tutkimuseettinen neuvottelukunta [8] määrittelee plagioinnin seuraavasti: "Plagioinnilla eli luvattomalla lainaamisella tarkoitetaan jonkun toisen julkituoman tutkimussuunnitelman, käsikirjoituksen, artikkelin tai muun tekstin tai sen osan, kuvallisen ilmaisun tai käännöksen esittämistä omana. Plagiointia on sekä suora että mukaillen tehty kopiointi." Ohjelmistoteollisuudessa tätä määritelmää täytyy laajentaa myös työn luvattomaan käyttöön. Luvattomassa käytössä ei suoranaisesti väitetä, että työ olisi tehty itse, vaan tarkoituksena on hyötyä toisen osapuolen resursseista ilman asianmukaista valtuutusta tai taloudellista korvausta.

Ohjelmakoodin plagiointi on helppoa ohjelmien ollessa suunniteltu uudelleenkäytettäväksi. Lisäksi ohjelmakoodi on luonteeltaan huomattavasti luonnollisia kieliä yksinkertaisempaa ja säännöllisempää, jolloin kopioitu materiaali sulautuu luonnollista kieltä helpommin kopioijan itse kirjoittamaan materiaaliin. Toisaalta samasta syystä kopioitu lähdekoodi on helpompi tunnistaa automaattisesti kuin luonnollisten kielten tapauksessa. Plagiointi on tyypillisimpiä syitä ohjelmistojen samankaltaisuuksien tarkasteluun ja siitä on selkeästi eniten julkaistuja kokemuksia. Toisaalta kokemukset ovat yleensä havaintoja löydetyistä kopiointitapauksista, eikä todellista plagiointitapausten määrää yleensä voida selvittää.

2.3 Samankaltaisuudet koulutusorganisaatioissa

Suomalaisissa koulutusorganisaatioissa pyritään yleensä puuttumaan ennakoivasti tilanteisiin, jotka saattaisivat johtaa plagiointiin, jolloin plagiointiin johtavat syyt täytyy tuntea. Whalen [13] mukaan ohjelmien samankaltaisuuksia voi ilmetä seuraavista syistä:

- sattuma
- yhteistyö

- plagiointi
- ulkopuolisesta lähteestä plagiointi.

Ohjelmat voivat olla sattumalta samankaltaisia lähes missä yhteydessä tahansa. Sattuman todennäköisyys vähenee kuitenkin ohjelmien monimutkaisuuden ja suuruuden kasvaessa. Toisaalta hyvin tarkat toiminnalliset vaatimukset lisäävät samankaltaisuuden todennäköisyyttä. Yliopistotason kursseilla saattaa olla jopa satoja opiskelijoita toteuttamassa samoja tehtäviä, joiden toiminnallisuus on tarkasti määritelty, jolloin työt muistuttavat aina jonkin verran toisiaan. Sattuman epätodennäköisyyttä voidaan myös käyttää raskauttavana todisteena esimerkiksi samalla tavalla väärin kirjoitettujen muuttujannimien avulla.

Yhteistyö on sattumasta seuraava vaihe. Yhteistyötapauksissa mahdollisen plagioinnin ja hyväksyttävän yhteistyön raja on vaikeasti hahmotettavissa, koska opiskelijoita usein rohkaistaan keskustelemaan keskenään ongelmista ja neuvomaan toisiaan ongelmatapauksissa. Tämä voi aiheuttaa opiskelijoissa hämmennystä tilanteissa, joissa opiskelija on omasta mielestään vain neuvonut toista opiskelijaa, mutta neuvonta on tehty esimerkiksi koodia sanelemalla, jolloin heikompi opiskelija ei tee itse työtään. Yhteistyö on kuitenkin usein yksipuolista siten, että heikompi opiskelija pyytää apua ongelman jo ratkaisseelta opiskelijalta.

Whalen [13] mukaan myös kurssihenkilökunta saattaa aiheuttaa ohjelmiin samankaltaisuuksia antamalla liian yksityiskohtaisia ja tarkkoja neuvoja. Tästä on myös kokemuksia Tampereen teknillisen yliopiston perusohjelmointikursseilla järjestettyjen koodausiltojen osalta. Koodausilloissa opiskelijat voivat tulla yhdessä tekemään ohjelmointitöitä yliopiston tiloihin erikseen mainittuina aikoina, jolloin paikalla on myös kurssihenkilökuntaa auttamassa mahdollisissa ongelmissa. Opiskelijat saattavat ymmärtää, että jos henkilökunta neuvoo opiskelijaa eteenpäin ja tämä ratkaisee ongelman, voi opiskelija jakaa ratkaisunsa vierustoverilleen. Toisaalta eri opiskelijoilla on usein hyvin samankaltaisia ongelmia, ja huoleton henkilökunnan edustaja saattaa huomaamattaan ohjata opiskelijoita samankaltaiseen ratkaisuun.

Varsinainen plagiointi tarkoittaa tilannetta, jossa opiskelija tietää tekevänsä väärin. Plagioinnissa opiskelija kopioi toisen opiskelijan työn ja mahdollisesti muokkaa sitä piilottaakseen kopioinnin tai sovittaakseen vaikean ohjelmakoodipätkän omaan ohjelmaansa. Joy ja Luck [6] jakavat plagioijat vielä kahteen ryhmään, taidoiltaan heikkoihin opiskelijoihin ja huonosti motivoituneisiin opiskelijoihin. Taidoiltaan heikoilla opiskelijoilla ohjelmointitaito ja ymmärrys kopioitavasta ohjelmasta ovat yleensä puutteellisia, jolloin kopioinnin naamiointikeinoina on pääasiassa ulkoisia muutoksia, jotka eivät muuta ohjelman rakennetta. Kokemuksen perusteella heikot opiskelijat saattavat esimerkiksi poistaa kaikki kommentit, koska ne ovat vapaampia ja yleensä

mukailevat alkuperäisen tekijän henkilökohtaista kirjoitustyyliä. Huonosti motivoituneet opiskelijat eivät ole välttämättä heikkoja, vaan opiskelijan ohjelmointitaito ja ymmärrys ohjelmoinnista voivat olla hyviä. Huonosti motivoitunut opiskelija pyrkii minimoimaan oman työnsä määrää. Tällainen opiskelija saattaa tehdä monimutkaisiakin muutoksia kopioinnin peittelemiseksi, jolloin kopioinnin havaitseminen on vaikeaa.

Whalen mukaan viimeinen ryhmä on ulkopuoliset lähteet, esimerkiksi aikaisempien vuosien opiskelijat. Erityisesti jos samoja ohjelmointitöitä uusiokäytetään vuosittain, opiskelijat saattavat pyytää neuvoa tai valmiita töitä vanhemmilta opiskelijoilta. Äärimmäistapauksissa opiskelija saattaa palkata täysin ulkopuolisen henkilön tekemään työn puolestaan, jolloin kopioinnin löytäminen on erittäin vaikeaa. Ulkopuolisena lähteenä voi toimia myös Internet, jossa on lukemattomia avoimen lähdekoodin ohjelmistoja jokaisen ladattavissa. Osa Internetissä olevasta materiaalista vaatii käyttäjältä rekisteröintiä, mikä vaikeuttaa kopiointien jäljittämistä.

2.4 Plagioinnin naamiointi

Plagiointia yritetään usein peitellä naamioimalla plagioituja osia sellaisiksi, että niiden alkuperää ei voitaisi jäljittää. Naamioinnista ei ohjelmistoteollisuuden puolella ole juuri todisteita tai esimerkkejä, koska suurimmassa osassa tapauksista kyse on lisensseistä ja niiden tulkinnasta, eikä varsinaisesti siitä, onko lähdekoodissa samankaltaisuuksia vai ei. Koulutusorganisaatioissa plagiointia on onnistuttu paljastamaan ja tunnetut naamiointimenetelmät on aiemmin löydettyjen plagiaattien perusteella johdettuja tai etsijöiden itsensä keksimiä. Seuraavaksi käsitellään koulutusorganisaatioissa tapahtuvaa naamiointia.

G. Whale [13] listaa naamiointimenetelmiksi kohdat 1-12. Kohta 13 liittyy erityisesti olio-ohjelmointiin.

1. Kommenttien muokkaaminen.
2. Tunnisteiden muokkaaminen.
3. Lausekkeiden operandien järjestyksen muokkaaminen.
4. Tietotyyppien vaihtamisen.
5. Lausekkeiden esitystavan vaihtaminen.
6. Tarpeettomien lausekkeiden lisääminen.
7. Lauseiden järjestyksen vaihtaminen.

8. Silmukkarakenteiden vaihtaminen.
9. Ehtorakenteiden muokkaaminen.
10. Funktiokutsujen vaihtaminen funktiorunkoon.
11. Lauseiden pilkkominen useampaan osaan.
12. Kopioidun ja itse tehdyn ohjelmakoodin yhdistäminen.
13. Rajapinnan kätkemän toteutuksen vaihtaminen.

Käytetty ohjelmointikieli vaikuttaa paljon siihen, kuinka helppoa eri naamiointien tekeminen on. Esimerkiksi C++ tarjoaa huomattavan määrän erilaisia keinoja ohjelmakoodin naamioimiseksi, josta järjestetään jopa kilpailuja [1]. Yleisesti muutokset voidaan jakaa kahteen ryhmään, kielellisiin ja rakenteellisiin muutoksiin.

Kielelliset muutokset tarkoittavat yksinkertaisia tekstinkorvausmenetelmiä. Kielellisiä muutoksia kykenee tekemään myös henkilö, jolla ei ole juurikaan tietoa naamioitavasta ohjelmasta tai ohjelmoinnista yleisesti. Kielelliset muutokset eivät yleensä muuta ohjelman rakennetta, vaan vaikuttavat pääosin vain ohjelmakoodin ulkoasuun. [6]

Rakenteelliset muutokset tarkoittavat ohjelman rakenteen muokkaamista. Rakenteelliset muutokset voivat vaikuttaa paljon ohjelman sisäiseen rakenteeseen ja toteutukseen, vaikka toiminnallisuus pysyisikin samana. Rakenteellisten muutosten havaitseminen on huomattavasti vaikeampaa kuin kielellisten muutosten. Ongelmaksi muodostuu erityisesti naamioidun ohjelman ja aidosti eri toteutuksen erottaminen. Rakenteellisten muutosten lisääminen lähentää järjestelmää aidosti uudeksi toteutukseksi. Rakenteellisten muutosten tekeminen vaatii yleensä syvää tietämystä ohjelmoinnista ja ratkaistavasta ongelmasta. [6]

Kielelliset ja rakenteelliset muutokset saattavat kuitenkin olla ulkoisesti hyvin lähellä toisiaan. Esimerkiksi ohjelmointikielissä, joissa on hyppykäsky **break**, voidaan **if**-lause vaihtaa **while**-lauseeksi ja sijoittaa silmukan viimeiseksi lauseeksi **break**. Muutos on yksinkertaista tekstinkorvausta, mutta se muuttaa ohjelman rakennetta huomattavasti kielen käsitteiden kannalta, vaikka loogisesti toiminnallisuus pysyykin samana.

Plagioidun materiaalin tunnistamiseksi naamioinnit pitäisi saada poistettua siten, että ohjelma vastaa alkuperäistä ohjelmakoodia. Toisaalta ohjelmointikursseilla saattaa olla satoja opiskelijoita toteuttamassa täsmälleen samaa tehtävänantoa, joten naamiointien tunnistaminen täytyy suunnitella huolellisesti, ettei samaa tavoitetta

[1] http://www.ioccc.org/

toteuttavat, täysin toisistaan riippumattomat ohjelmakoodit vaikuttaisi plagioidulta materiaalilta. Samankaltaisuuksien tunnistamiseen useampia erilaisia lähestymistapoja, joita käsitellään seuraavaksi.

3. PLAGIOINNIN TUNNISTAMINEN

Plagioinnin tunnistaminen tarkoittaa jotain järjestelmällistä tapaa seuloa plagiaatteja lähdekooditiedostoista. Käytännössä samankaltaisuuksien etsinnän suorittaminen käsin muodostuu liian työlääksi jo muutamalla kymmenellä lähdekooditiedostolla, joten etsintä yleensä etsintä suoritetaan koneellisesti ja vain koneellisen etsinnän tulokset tarkistetaan käsin. Tässä luvussa kuvataan plagioinnin tunnistamisen prosessi sekä käsitellään yksityiskohtaisemmin prosessin toteuttavia menetelmiä.

3.1 Vaiheet

Culwin ja Lancaster [3] jakavat tunnistusprosessin neljään vaiheeseen: materiaalin keräykseen, analyysiin, varmistukseen ja tutkintaan. Tunnistusprosessin suorittajia voi olla yksi tai useampia, mutta rooleista keskeisin on varmistusvaiheen suorittaja, josta käytetään jatkossa termiä *vertailun suorittaja*, koska käytännössä varmistusta johtaa yleensä se henkilö, joka käyttää plagiaatintunnistusohjelmaa.

Materiaalin keräämisellä tarkoitetaan rajatun vertailujoukon muodostamista eri lähteistä. Ideaalitapauksessa vertailujoukko muodostettaisiin koko toteutusavaruudesta, jolloin vertailu voisi ottaa huomioon kaikki mahdolliset lähteet. Koko toteutusavaruuden käyttö olisi kuitenkin käytännössä mahdotonta rajallisten laskentaresurssien ja lähdekoodien saavutettavuuden vuoksi, joten vertailujoukko koostetaan tiukasti rajatusta lähteiden joukosta, joka esimerkiksi TTY:n ohjelmointikursseilla tarkoittaa kurssien automaattista palautustenhallintajärjestelmää. Tyypillisesti suurin osa kopioinneista tapahtuu saman tai aikaisempien vuosien opiskelijoilta, joiden tekemät ohjelmat löytyvät usein paikallisesta arkistosta. Ongelmallisinta on ulkoisten henkilöiden palkkaaminen työtä varten, jolloin plagioinnin kohteena olevan lähdekoodin saavuttaminen on yleensä mahdotonta.

Analyysivaiheessa suoritetaan koneellinen vertailu. Tarkastettavina olevista tiedostoista pyritään löytämään kopioituja osia, joita on mahdollisesti yritetty naamioida. Analyysivaiheen tulokset kuuluvat aina johonkin neljästä luokasta, oikea positiivinen, väärä positiivinen, väärä negatiivinen ja oikea negatiivinen. Ideaalitapauksessa kaikki tulokset kuuluisivat ryhmiin oikea positiivinen tai oikea negatiivinen,

jolloin analyysityökalu löytäisi aina kaikki todelliset kopiointitapaukset eikä havaitsisi epäilyttäviä määriä samankaltaisuuksia sellaisissa tapauksissa, joissa kopiointia ei ole tapahtunut. Käytännössä sattumalta samanlaiset lähdekoodit takaavat, että ideaalitapaus on lähes mahdoton saavuttaa täydellisesti. Väärä positiivinen ja väärä negatiivinen ovat analyysityökalun puutteita. Väärät positiiviset ovat sellaisia, jotka eivät ole kopioita, mutta analyysityökalun mukaan ne sisältävät epäilyttävän määrän samankaltaisuuksia. Väärä negatiivinen tarkoittaa, että kopioitua materiaalia ei tunnisteta. Väärien positiivisten määrä pitäisi saada karsittua minimiin, jotta vertailua voidaan pitää hyödyllisenä. Väärät positiiviset aiheuttavat tarpeetonta työtä varmistusvaiheessa ja mahdollistavat tilanteen, jossa syytöntä epäillään plagioinnista.

Varmistusvaiheessa pyritään karsimaan väärät positiiviset tulosjoukosta ennen yhteydenottoa tapauksen osapuoliin. Varmistusvaihe tehdään yleensä käsin ja siinä etsitään analyysivaiheen tulosta vahvistavia todisteita kopioinnista. Varmistusvaihetta käsitellään tarkemmin kohdassa 3.3.

Tutkintavaiheessa tapauksen osapuolet kutsutaan erikseen kuultaviksi. Kuulemisessa vertailun suorittaja esittää keräämänsä todisteet eri osapuolille ja plagioinnista epäillyt voivat puolustautua tai myöntää syyllisyytensä. Tarpeen mukaan tapausta voidaan käsitellä useamman kerran. Käsittelyn jälkeen julistetaan seuraus, joka voi olla vapauttava, langettava tai jotain siltä väliltä. Langettava seuraus riippuu sen organisaation, jossa plagiointiin on syyllistytty, säännöistä ja toimintatavoista, mutta tyypillisesti plagiointiin syyllistyneet menettävät vähimmäistapauksessa kaikki niiden kurssien suoritusmerkinnät, jotka on hankittu vilpillisin keinoin. Lievissä tapauksissa, esimerkiksi jos opiskelijat ovat tehneet enemmän yhteistyötä kuin on sopivaa, voidaan opiskelijoilta vaatia lisätöitä plagioinnin korvaamiseksi.

3.2 Analyysivaiheen menetelmät

Automaattisen tunnistuksen tärkeimmät ratkaistavat ongelmat ovat lähdekoodin esitysmuoto tunnistuksen aikana ja menetelmä, jolla käytettyä esitysmuotoa tulkitsemalla vertailtavat lähdekoodit voidaan järjestää siten, että mahdolliset väärinkäytökset voidaan löytää. Ohjelman esitysmuoto on kieli, joka kuvaa alkuperäistä lähdekoodia. Esitysmuodon kielen ei tarvitse olla sama kuin alkuperäisen lähdekoodin, ja usein tarkoituksena on muuttaa alkuperäinen lähdekoodi yleistetyksi malliksi, jolloin ulkomuodolliset yksityiskohdat saadaan poistettua. Esitysmuodon tulkitsemiseksi tarvitaan vertailukeino, jolla eri lähdekoodit voidaan asettaa järjestykseen niiden samankaltaisuuden mukaan.

3.2.1 Esitysmuoto

Esitysmuototyypit voidaan jakaa karkeasti kahteen osaan: Ominaisuuksien laskentaan ja ohjelman kääntämiseen toiseksi kieleksi. Ominaisuuksien laskennassa ohjelmakoodi esitetään ohjelmointikielen osien lukumäärinä. Osasina voi olla esimerkiksi operaattorit, operandit, funktiot, funktiokutsut ja tiedostojen rivimäärät. Ominaisuuksien laskennassa kaikki alkuperäisen ohjelman rakenne hukataan täysin, mutta hyvänä puolena ohjelman esitysmuoto vie erittäin vähän tilaa. Ominaisuuksien laskentaa pidetään yleisesti helposti huijattavana menetelmänä. Pohjimmiltaan myös kääntämiseen perustuvat vertailumenetelmät perustuvat kuitenkin ominaisuuksien suhteelliseen laskentaan käännösten jälkeen, jolloin pyritään erottamaan suuresta massasta poikkeavat yhteneväisyydet.

Ominaisuuksien laskenta toimii silloin, kun kopioituun lähdekoodiin ei ole tehty juurikaan muutoksia. Suurin osa kopioijista pyrkii kuitenkin peittelemään kopiointia muuttamalla ohjelmakoodia toisen näköiseksi. Vastatoimena kopioinnin peittelylle voidaan lähdekoodin esitysmuoto muuttaa edelleen toisenlaiseksi, yleistetyksi malliksi. Yleistys perustuu symboleiksi muuttamiseen (engl. tokenization). Yleistämisen lopputuloksena voi kuitenkin syntyä yleistysmenetelmästä ja -asteesta riippuen erilaisia malleja, kuten graafeja, jäsennyspuita, semanttisia symbolijonoja ja puhtaita symbolijonoja. Graafeissa kuvataan ohjelma tilakoneena, joissa tiloina voivat olla esimerkiksi kaikki erilliset ohjelmalohkot. Jäsennyspuussa ohjelmakoodi käännetään symboleiksi ja jäsennetään, jolloin esimerkiksi lausekkeiden suoritusjärjestys säilyy. Semanttisella symbolijonolla tarkoitetaan sellaista symbolijonoa, jossa viittaukset samaan tunnisteeseen voidaan yhä tunnistaa, esimerkiksi kaikki funktiokutsut tai muuttujaviittaukset osataan yhä yhdistää. Malli voi olla myös puhtaasti kielen abstrakteista käsitteistä muodostuva symbolijono, joka on ainoa edellä mainituista ohjelman malleista, joka ei vaadi kokonaista ohjelmaa toimiakseen oikein. Puhdasta symbolijonoa käsitellään tarkemmin alakohdassa 3.2.3. Yksinkertaisin malli on jättää lähdekoodi sanasta sanaan alkuperäiseksi ilman minkäänlaista esitysmuodon muokkausta, jolloin vain suorat kopiot voidaan löytää helposti.

Yleistämisen tarkoituksena on luoda sellainen malli, jossa alkuperäinen ja plagioijan muokkaama ohjelmakoodi näyttäisivät yleistyksen jälkeen samoilta siten, että kenenkään muun itse kirjoittama ohjelmakoodi ei näyttäisi samalta. Yleistetyn mallin etuna on se, että yksinkertaiset tekstinkorvaukseen perustuvat naamiointikeinot saadaan palautettua alkuperäisen näköisiksi.

3.2.2 Vertailumenetelmä

Yleistäminen ei havaitse ohjelman rakenteen muuttamiseen perustuvia naamiointikeinoja, vaan yleistettyyn muotoon muunnettuja lähdekooditiedostoja täytyy verrata toisiinsa jollain menetelmällä. Vertailu suoritetaan vertailemalla jokaista *vertailualkiota* kaikkiin muihin vertailujoukon alkioihin. Vertailualkio voi olla toteutuksesta riippuen yksittäinen tiedosto tai kokonainen ohjelma. Myös pienemmät ohjelman osat voisivat olla vertailualkiona, mutta ongelmaksi muodostuu alkioihin jako. Tässä työssä vertailualkiona on käytetty yhtä tiedostoa. Vertailu tuottaa vertailun *tulosjoukon*, joka muodostuu vertailujoukon alkiopareista. Tulosjoukkoon voidaan myös tuoda *alkiorinkejä*, jolloin useamman vertailualkion on havaittu muistuttavan toisiaan. Yksinkertaisuuden vuoksi tässä työssä käsitellään alkiopareja, koska alkioringit voidaan aina pilkkoa pareiksi.

Tulosjoukko muodostetaan vertailujoukosta siten, että kaikki vertailujoukon alkioparit järjestetään jonkin samankaltaisuutta arvioivan algoritmin avulla samankaltaisuusjärjestykseen, ja tietyn samankaltaisuuden asteen ylittävät parit otetaan mukaan tulosjoukkoon. Samankaltaisuutta esitetään usein prosentuaalisesti, mutta algoritmien moninaisuudesta johtuen pelkkä prosenttimerkintä ei yleensä yksistään kerro todellisesta samankaltaisuudesta, vaan prosenttilukumäärän merkitys täytyy tuntea. Yleisesti prosentuaalisella esityksellä voidaan kuitenkin antaa yksinkertainen ja helposti havainnollistava merkintä vertaillun ohjelmaparin samankaltaisuudesta.

Ohjelmaparit voidaan järjestää erilaisilla samankaltaisuutta arvioivilla algoritmeilla, joista jokaisella on omat hyvät ja huonot puolensa. Huonojen puolien perustana on deterministisen algoritmien tapauksessa se, että jos algoritmi tunnetaan etukäteen, voi plagioija yrittää kehittää sellaisia naamiointeja, joita kyseinen algoritmi ei löydä. Algoritmien heikkouksia vastaan hyökkäämistä voidaan vaikeuttaa käyttämällä useamman erilaisen algoritmin yhdistelmiä tai valitsemalla algoritmi tai algoritmit jokaiselle työkalun ajokerralle satunnaisesti. Satunnaisen yhdistelemisen vaarana on kuitenkin valita sellaiset algoritmit, joilla on sama heikkous. Algoritmeja valittaessa täytyy myös huolehtia, etteivät ne ole liian herkkiä samankaltaisuuksien tunnistamisessa. Liian herkät algoritmit lisäävät väärien positiivisten hälytysten määrää ja vähentävät siten niiden luotettavaa karsintaa varmistusvaiheessa.

Algoritmeja on lukuisia, joista tässä työssä esitellään kaksi mahdollista vaihtoehtoa, pisimpien yhteisten osajonojen laskenta ja yhteisten rivien laskenta. Pisimpien yhteisten osamerkkijonojen etsintä on hidasta ja altis koodin järjestyksen muutoksille, mutta sitä on vaikea huijata esimerkiksi lisäämällä ylimääräisiä lauseita kopioitujen

osien keskelle. Pisimmän yhteisen osajonon algoritmia käsitellään tarkemmin kohdassa 4.3. Toinen vaihtoehto on laskea yhteisten symbolijonorivien määriä, jolloin lähdekoodin järjestyksellä ei enää ole merkitystä. Toisaalta tämä ratkaisu ei löydä välttämättä osittaisia täysin suoria kopioita, jos riittävän suuri osa tiedostoista on erilaista. Rivien käyttäminen vertailtavina yksikköinä ei aiheuta ongelmia, koska muunnoksista voidaan tehdä sellaisia, että rivit jaetaan aina tietynlaisista kohdista riippumatta siitä millainen alkuperäinen lähdekoodi on. Yhteisten rivien algoritmia käsitellään tarkemmin kohdassa 4.2

3.2.3 Käsitekieli

Käsitekielellä tarkoitetaan ohjelmointikielen käsitteitä ja rakenteita esittävää kieltä. Käsitekieli perustuu Lofti A. Zadehn [14] teoriaan lingvistisistä muuttujista. Perusajatus on, että ohjelmointikielen komponentit kuvataan yleistettyinä symboleina. Esimerkiksi kaikkia tunnisteita voidaan kuvata samalla lingvistisen muuttujan arvolla.

Käsitekieli ei aseta vaatimuksia käytetyille muunnoksille, vaan käsitekielen ainoa varsinainen vaatimus on olla deterministinen kuvaus lähdemerkistöstä käsitekielen merkistöön. Käytännössä tämä mahdollistaa käsitekielen dynaamisen valitsemisen kielestä riippuen, jolloin kielen rakennetta ei tarvitse pakottaa kiinteään muottiin.

Käsitekielen symbolien valinta tulisi tehdä niin, että tuotettu symbolijono vastaisi mahdollisimman hyvin ohjelman rakennetta, jota on tyypillisesti vaikea muuttaa pienehköillä muunnoksilla.

Esimerkkinä valtavirran ohjelmointikielessä on muutamia yhteisiä rakenteita. Yleisimmät käsitteet ovat tunniste I, varattu sana V ja operaattori. Kielestä riippuen operaattori kannattaa yleensä jakaa useampaan, C-sukuisten kielten tapauksessa kolmeen osaan: sijoitusoperaattoreihin A, käsittelyoperaattoreihin M ja muihin operaattoreihin O. Operaattorien ja tunnisteiden muodostamat lausekkeet voidaan kuvata symbolilla L. Käsitekielen perusaakkosto T on tässä tapauksessa

$$T(Kieli) = \{'V', 'I', 'L', 'O', 'A', 'M'\}$$

Näiden lisäksi tietyt ohjelmakoodia jäsentävät merkit, kuten ryhmityssulkeet, voidaan jättää mukaan sellaisenaan, koska ne pysyvät samoina riippumatta koodin kirjoittajasta.

Käsitekielelle muuntaminen riippuu paljon käytetystä ohjelmointikielestä. Kääntäminen on suurelta osin mahdollista toteuttaa säännöllisten lausekkeiden avulla, joskin muunnoksen tekeminen oikealla kääntäjällä olisi tehokkaampi ratkaisu. Säännöllisten lausekkeiden käyttö mahdollistaa kuitenkin uusien kielten ja muunnosten helpon lisäämisen. Jokaiselle ohjelmointikielelle voidaan kirjoittaa sääntöjoukot, joiden perusteella muutoksia tehdään. Lopputuloksena syntynyttä symbolijonoa voidaan vertailla toisiin symbolijonoihin riippumatta alkuperäisestä kielestä. Jos eri kielistä tuotettuja symbolijonoja vertaillaan keskenään, riippuu vertailun mielekkyys käsitekielten eroista. Säännöllisillä lausekkeilla muuntamisessa lausekkeiden suoritusjärjestyksellä voi olla merkitystä muunnoksesta riippuen, joten muunnokset täytyy suorittaa järjestyksessä.

```
unsigned int f( unsigned int n ){     I(I)
    if( n <= 2 ) {                    V(L)
        if( n == 0 ){                 V(L)
            return 0;                 VI;
        }
        return 1;                     VI;
    }
    return f(n - 1) + f(n - 2);       VI(L)OI(L);
}
```

Kuva 3.1: Esimerkki lähdekoodista ja sitä vastaavasta käsitekielisestä symbolijonosta

Kuvassa 3.1 on näytetty yksi tapa muuttaa C++-kielellä kirjoitettu ohjelmakoodi käsitekieliseksi symbolijonoksi. Tyhjät rivit on jätetty muunnettuun versioon, jotta olisi helpompi havaita, mikä rivi on muunnettu miksi symbolijonoriviksi. Tuotetusta symbolijonosta voidaan havaita, että käytetyssä muunnosoperaatiossa on yhdistetty peräkkäiset samat symbolit yhdeksi. Mikäli näin ei olisi tehty, olisi muunnoksen ensimmäinen rivi muotoa $III(III)$, joka olisi selkeästi tarkempi muunnos alkuperäisestä. Tätä voi kuitenkin huijata helposti poistamalla paluuarvon ja parametrin n tyyppikuvauksesta sanan **int**, jolloin tuotettu symbolijono muuttuisi muotoon $II(II)$, vaikka ohjelma pysyisi muuten täysin samanlaisena.

3.3 Plagioinnin varmistaminen

Automaattinen tunnistus ei yleensä voi tehdä varmoja päätelmiä siitä, ovatko samankaltaiset ohjelmat todella plagiaatteja. Automaattisen tunnistuksen tavoitteena on seuloa suuri määrä informaatiota siten, että lopullinen varmistus voidaan tehdä järkevässä ajassa käsin.

Käsitekielen muunnoksissa yritetään häivyttää kaikki ohjelman kannalta merkityksettömät osat kuten kommentit, rivinvaihdot ja tyhjät rivit. Tämä voi olla vertailun

kannalta hankalaa kun tarkastellaan lyhyitä ja yksinkertaisia ohjelmia, joissa ohjelman toteutuskoodia on hyvin vähän, jolloin vaarana on väärien positiivisten tulosten antaminen. Käsin tehtävässä varmistusvaiheessa voidaan kuitenkin ottaa huomioon myös seulontavaiheessa pois häivytetyt erikoispiirteet.

Plagiointia ei voida aina todistaa kiistattomasti ohjelmakoodeja tarkastelemalla. Käsin tehdyssä varmistuksessa voidaan kuitenkin löytää piirteitä, joiden toistuessa plagioinnin todennäköisyys kasvaa huomattavaksi. Esimerkkejä plagiointiin viittaavista piirteistä ovat esimerkiksi ylimääräiset merkit samoissa paikoissa ohjelmakoodia, samanlaiset kirjoitusvirheet, hyvin erikoiset ratkaisut ja samat virheet ohjelman toiminnassa. Plagiaatiksi toteamista käsitellään tarkemmin luvussa seuraavaksi.

3.4 Plagiaatin kriteerit

Samankaltaisuus ei välttämättä tarkoita plagiaattia. Siksi on tärkeää määritellä sellaiset ehdot, joilla samankaltaisista ohjelmista voidaan erotella todelliset plagiaatit. Samankaltaisuuden määritelmän epämääräisyyden vuoksi on vaikeaa määritellä sellaisia ehtoja, jotka eivät vetäisi jossain tilanteessa plagiaatin rajaa juuri sellaiseen kohtaan, jossa plagiaatteja vielä esiintyy. Plagiaatiksi tunnistettavan ehdot jakautuvat useampaan erilaiseen tyyppiin, koska ehtojen arvioija ei ole jokaisessa tunnistusprosessin vaiheessa sama. Plagiaatin *esiehdot* ovat sellaisia vaatimuksia, joiden täyttyminen on pakollista plagiaatin tunnistamiseksi. Termillä *pari* tarkoitetaan tässä kahta vertailujoukon alkiota. Plagiaatin esiehtoja ovat seuraavat:

Saavutettavuus. Parin molempien osapuolten täytyy olla vertailujoukossa. Saavutettavuus on helppo tarkistaa, koska pari voidaan joko löytää tai olla löytämättä vertailujoukosta. Saavutettavuutta on kuitenkin mahdotonta taata, ellei vertailun suorittajalla ei ole tietoa kaikista olemassa olevista yksiköistä.

Tulkittavuus. Parin molempien osapuolten pitää olla vertailutyökalun tulkittavissa siten, että vertailutyökalu voi antaa parille samankaltaisuusarvon. Tulkittavuusehto ei välttämättä päde, jos toinen parin osapuolista on esimerkiksi kirjoitettu sellaisella kielellä, jota vertailutyökalu ei ymmärrä. Tulkittavuusehto on triviaalisti todettavissa, jos saavutettavuus täyttyy.

Samankaltaisuus. Parin pitää täyttää sellainen samankaltaisuuden raja, jonka perusteella vertailutyökalu sijoittaa parin epäiltyjen listalle. Käytännössä aina vertailutyökalun tulosjoukkoa täytyy rajata jollain keinotekoisella rajalla, koska muuten tulosjoukko kasvaa niin suureksi, ettei sitä pysty järkevästi

käsittelemään. Samankaltaisuuden rajaa pystyy yleensä myös säätämään, koska sopiva raja riippuu usein vertailtavan aineiston luonteesta. Samankaltaisuuden raja esiehtona ei ota kantaa sisältöön, vaan pelkästään keinotekoisesti valittuun rajaan.

Näkyvyys. Parin pitää sijoittua listalla sellaiseen kohtaan, joka päätyy käyttäjän tarkasteltavaksi. On mahdollista, että epäiltyjen lista kasvaa niin suureksi, että vertailutyökalu joutuu rajaamaan näytettäviä pareja, vaikka ne olisivatkin päätyneet listalle. Jos vertailutyökalusta voi poistaa rajoituksen, näkyvyysehto on helppo täyttää. Näkyvyysehdosta tulee rajoittava tekijä silloin, kun samankaltaisuusraja on valittu huonosti tai vertailutyökalu rajoittaa näkyvän tulosjoukon kokoa liian helposti.

Esiehtojen täyttymisen jälkeen vertailun suorittaja arvioi muiden ehtojen toteutumista. Tulosjoukon suuruus vaikuttaa vertailun suorittajan tehtävään paljon siten, että jos tulosjoukko on riittävän pieni, voi vertailun suorittaja käydä läpi koko epäiltyjen listan. Samankaltaisuuden raja on kuitenkin hyvin vaikea valita siten, että varmasti kaikki plagiaatiksi mahdollisesti tunnistettavat päätyvät tulosjoukkoon, mutta ei yhtään ylimääräisiä. Vertailun suorittaja ei suorita ehtoja sellaisessa mielessä, että ne voivat saada arvon tosi tai epätosi, vaan epäiltyjen listaa tarkistettaessa vertailun suorittaja etsii todisteita plagioinnista. Todisteet voivat olla *riittäviä*, jolloin todiste yksinään riittää päätökseen plagiointiepäilystä. Riittävät todisteet ovat harvinaisia ja yleensä seurauksia plagioijan huolimattomuudesta, kuten alkuperäisen tekijän nimen jättäminen ohjelmakoodin kommentteihin osoittamaan oikeaa tekijää. *Aihetodisteet* eivät suoraan osoita syyllisyyttä, mutta lisäävät sen todennäköisyyttä, ja riittävän monen aihetodisteenkin perusteella voidaan tehdä päätelmä todennäköisestä syyllistymisestä plagiointiin. Todisteiden vahvuuksissa voi olla myös huomattavia eroja. Todisteen vahvuutta mitataan sen perusteluvoimassa, jolloin vahva todiste on sellainen, jonka avulla kolmas osapuoli voidaan vakuuttaa epäilyksen aiheellisuudesta. Plagiointiepäilyyn vaadittavien aihetodisteiden määrää ja vahvuutta on mahdoton määritellä etukäteen, jolloin jokainen tapaus täytyy arvioida tapauskohtaisesti. Luonnollisesti vahvoja todisteita tulisi olla niin paljon, että niillä voi perustella vakuuttavasti epäilystä plagioinnista. Todisteet ovat yleensä hyvin tapauskohtaisia eikä kaikkien mahdollisten todisteiden listaaminen etukäteen ole mahdollista. Vertailun suorittaja voi kuitenkin etsiä seuraavantyyppisiä todisteita

Nimimerkkiristiriita. Joihinkin ohjelmointikäytäntöihin kuuluu, että tekijän nimi, käyttäjätunnus, sähköposti tai opiskelijanumero lisätään jokaiseen kooditiedostoon otsikkotietoihin tai opiskelijoita voidaan ohjeistaa kirjoittamaan jokainen ohjelma niin, että se tulostaa tekijän nimen aina ensimmäiseksi. Joskus plagioijan huolimattomuudesta johtuen voi alkuperäisen tekijän

nimi jäädä ohjelmakoodiin, joka yleensä katsotaan riittäväksi todisteeksi.

Jaettu virhe. Jaettu virhe on jokin parin molemmissa osapuolissa esiintyvä ohjelmointivirhe. Virheen laatu määrittelee todisteen vahvuuden. Virhetodiste voi olla hyvin vahva, jos virhe on sellainen, joka esiintyy vain hyvin harvoissa yksilöissä tai joka ei näy esimerkiksi kurssihenkilökunnan tuottamassa testimateriaalissa.

Jaettu erikoisuus. Jaetulle erikoisuudella tarkoitetaan parin molemmista osapuolista löytyvää yhteistä ylimääräistä koodia, ratkaistavalle tehtävälle erikoista esitystä tai ohjelmoijan oletetuille taidoille epätyypillistä ratkaisua. Jaettu erikoisuus voi olla kohtalaisen vahva todiste riippuen erikoisuuden laadusta.

Kompleksisuusperiaate. Kompleksisuusperiaate perustuu siihen, että mikäli ohjelman monimutkaiset ja paljon entropiaa sisältävät osat muistuttavat toisiaan huomattavasti, mutta esimerkiksi yksinkertaiset alustus- ja tulostusoperaatiot ovat epätavallisen erinäköisiä, on syytä epäillä plagioinnin peittelyä. Ohjelmointitaidoiltaan heikkojen henkilön on vaikea muuttaa monimutkaisia osia ohjelmasta niin, että ne vielä toimivat oikein, koska usein juuri vaikeat osat ovat olleet motivaationa plagiointiin.

Jaettu kommentti. Plagioija saattaa jättää myös kommentteja muuttamatta, jolloin ohjelmointikielen asettamat rajoitteet tekstin monimuotoisuudelle eivät enää päde. Erityisesti pitkät sanasta sanaan samat kommentit voivat olla vahvoja todisteita.

Tekstin muotoilu. Tekstin muotoilun yksityiskohdat ovat yleensä pieniä aihetodisteita, mutta saattavat toistuessaan osoittaa syyllistymistä niin kutsuttuun leikkaa ja liimaa -kopiointiin. Muotoiluseikkoja on erittäin paljon ja niitä käytetään yleensä vahvistamaan aiempia todisteita. Muotoilutodisteita voivat olla esimerkiksi ohjeistuksesta poikkeavat ohjelmointityyliseikat kuten tyhjien merkkien tai lohkosulkeiden sijainti, tulostuslauseiden tai useamman rivin lausekkeiden samanlainen rivittäminen ja muotoilu. Näkymättömät merkit voivat olla vahvoja todisteita, esimerkiksi ylimääräinen välilyönti aina keskenään samanlaisten lauseiden lopussa.

Jaettu ohjelmakoodi. Ohjelmakoodin jakaminen sellaisenaan on yksinkertaisin todiste, koska tulkintaa samankaltaisuudesta ei tarvitse tehdä. Jaettu ohjelmakoodi ei kuitenkaan ole riittävä todiste kopioinnista, vaan tarkastajan on otettava huomioon ohjelmakoodin muoto, monimutkaisuus ja asiayhteys. Osa ohjelmakoodista voi olla tarkoituksella samanlaista, kuten esimerkiksi silmukoiden indeksimuuttujien nimeäminen tai yksinkertaisten tietorakenteiden

alustustoimenpiteet. Jaettua ohjelmakoodia voi olla hyvin vaihtelevassa määrin, muutamista samanlaisista muuttujien nimeämisestä koko ohjelmakoodin täydelliseen kopioon.

Riittävän todistejoukon keräämiselle ei voi asettaa kiinteää rajaa, vaan jokainen tapaus täytyy käsitellä tapauskohtaisesti. Riittävyyden arvioinnista vastaa vertailun suorittaja, jolloin riittävä todistejoukko voi vaihdella suorittavan henkilön mukaan.

```
bool onko_pilalla ( const int lampotila)
{
   if (lampotila < 20 or lampotila > 25)
   {
      return true;
   }
   return false;
}
```

Kuva 3.2: Esimerkkikoodi erikoisesta const-määreen käytöstä todellisessa kopiointitapauksessa

Kuvassa 3.2 on esimerkki todellisesta kopiointitapauksesta, joka automaattisessa tarkistuksessa nousi epäilyttävien listalle ja käsin tehdyssä tarkastelussa epäilykset vahvistuivat entisestään **const**-määreen käytöstä paikassa, jossa sen käyttö oli epätavallista. On mahdollista, että kirjoittajilta on jäänyt epähuomiossa viitteen merkki kirjoittamatta, mutta on epätodennäköistä että kaksi opiskelijaa kirjoittaisi täsmälleen samanlaisen funktion, jossa on täsmälleen sama kirjoitusvirhe.

4. VERTAILUALGORITMIT

Vertailua varten tulee valita sopivat mittarit, jotka parhaiten vastaavat samankaltaisuutta. Paras tulos saavutetaan yhdistelemällä useampia mittareita, jolloin yksittäisen mittarin ei tarvitse täyttää kaikkia vertailulle asetettuja vaatimuksia. Käsitekieleksi tulkinnan jälkeen tiedostoja vertaillaan vertailualgoritmeilla, joilla yritetään löytää plagiaatteja käsitekielisistä symbolijonoista. Tässä luvussa käsitellään yksityiskohtaisesti pisimmän yhteisen osajonon algoritmi ja yhteisten rivien algoritmi.

4.1 Pienin vertailuyksikkö

Vertailualgoritmit käsittelevät yleensä kerrallaan yhtä vertailujoukon alkioparia. Vertailualkio voidaan jakaa monella erilaisella tavalla vertailtaviin yksikköihin, esimerkiksi vertailemalla jokaista symbolia erikseen tai muodostamalla useamman merkin mittaisia symboliryhmiä. Tässä työssä esitellyt ja toteutetut algoritmit käyttävät yksikkönä *riviä*, joka ei kuitenkaan tarkoita välttämättä todellista ohjelmakoodiin kirjoitettua riviä, vaan symboleista muodostettuja ryhmiä, jotka on tiettyjen sääntöjen mukaan jaettu riveihin. Yksityiskohtaisia riveihin jakosääntöjä ei julkisteta tässä työssä, jotta niitä ei voitaisi tulevaisuudessa käyttää hyödyksi.

Vertailuyksikön koko vaikuttaa vertailun herkkyyteen. Pienet vertailuyksiköt lisäävät algoritmien ajankulutusta ja herkkyyttä, ja suuret nopeuttavat algoritmien suoritusta ja vähentävät herkkyyttä. Rivien käyttäminen vertailuyksikkönä tarjoaa mahdollisuuden vaihtelevan mittaiseen yksikön pituuteen, jolla pyritään säilyttämään algoritmin suorituskyky ja toisaalta pitämään herkkyys sopivalla tasolla. Rivien jakosäännöt ovat tässä suhteessa ratkaisevia algoritmin toiminnan kannalta. Valittu käsitekieli ja muunnokset vaikuttavat olennaisesti siihen, kuinka paljon lähdekoodin kirjoittajan tekemät muutokset vaikuttavat vertailuyksiköiden muodostukseen. Rivin pituus voidaan myös määrätä vakioksi, jolloin todellisilla rivinvaihdoilla on vielä vähemmän merkitystä, mutta käytettyä pituutta voidaan helposti huijata lisäämällä sopiva määrä ylimääräisiä kielen rakenteita vertailualkion alkuun. Jokaiselle riville voidaan antaa yksikäsitteinen tunniste, jolloin rivit supistuvat yhden kokonaisluvun kokoisiksi ja niiden vertailusta tulee nopeaa.

4.2 Yhteiset rivit

Yhteisten rivien laskenta kahdesta tiedostosta riippumatta rivien järjestyksestä kertoo hyvin, kuinka paljon tiedostoista on yhteistä. Toisaalta rivien järjestyksellä on ohjelman rakenteen kannalta huomattavasti merkitystä, eikä pelkkä tilastollinen määrä kerro rakenteesta. Kokemuksen perusteella kuitenkin yhteisten rivien laskenta toimii melko tarkasti, jos ohjelmat ovat pieniä tai ne on jaettu kovin samanlaisiin moduuleihin.

Yhteisten rivien laskennalla voidaan torjua erityisesti ohjelmakoodin järjestyksen muuttamista plagioinnin peittämiseksi. Kun rivien järjestyksellä ei ole merkitystä, ei ole myöskään väliä missä järjestyksessä rivit ovat alkuperäisessä tiedostossa. Yhteisten rivien laskenta on myös vahvoilla yleisen samankaltaisuuden ja kokonaisten tiedostojen kopioinnin löytämisessä.

Yhteisten rivien laskennan heikkoutena on tunnistaa osakopiointeja. Esimerkiksi kahdesta tuhannen rivin tiedostosta voi olla 500 riviä täysin suoraa kopiota, joka ei kuitenkaan näy suhteellisesti merkittävänä samankaltaisuutena tiedostoja vertailtaessa. Tällainen tiedosto pääsee helposti ohi seulan.

Jotta yhteisten rivien laskenta onnistuisi mahdollisimman helposti, määritellään tiedoston f_p sisällön muodostama monijoukko järjestettyinä pareina $\langle x, i\rangle$ siten, että x esittää riviä ja $i < 0$ rivin x esiintymisten määrää kyseisessä tiedostossa. Tällöin tiedostojen f_m ja f_n leikkauksen muodostama joukko voidaan muotoilla seuraavasti:

$$f_m \cap f_n = \{\langle x, min(i, j)\rangle \mid \forall x : \langle x, i\rangle \in f_m \wedge \langle x, j\rangle \in f_n\} \tag{4.1}$$

Yhtälön 4.1 avulla voidaan tiedostojen f_m ja f_n yhteiset rivit esittää seuraavalla tavalla:

$$yht(f_m, f_n) = \sum_{\langle x,i\rangle \in f_m \cap f_n} i \tag{4.2}$$

Tiedostojen määritteleminen järjestettyinä pareina mahdollistaa yhteisten rivien laskennan toteutuksen tehokkaasti. Yhteisten rivien laskennan toteutus on esitetty algoritmissa 4.1. Tiedostojen f_m ja f_n muodostamista parijoukoista muodostetaan järjestetyt taulukot A ja B. Yksittäistä paria käsitellään valitsimilla *line* ja *count*, joista ensimmäinen palauttaa parin esittämän rivin ja jälkimmäinen rivin esiintymismäärän.

```
1: procedure CommonLines(A,B)
2:     m ← length(A)
3:     n ← length(B)
4:     i ← 1
5:     j ← 1
6:     k ← 0
7:     while i ≤ m ∧ j ≤ n do
8:         if A[i].line < B[j].line then
9:             i ← i + 1
10:        else
11:            if B[j].line < A[i].line then
12:                j ← j + 1
13:            else
14:                k ← k + min(A[i].count, B[j].count)
15:                i ← i + 1
16:                j ← j + 1
17:            end if
18:        end if
19:    end while
20: end procedure
```

Algoritmi 4.1 Algoritmi yhteisten rivien laskentaan.

Yhteisten rivien laskenta on nopeaa algoritmilla 4.1 ajankulutuksen ollessa kertaluokassa $O(n)$, jossa n on vertailtavana olevista tiedostoista pidemmän pituus.

4.3 Pisin yhteinen osajono

Pisin yhteinen osajono (engl. longest common subsequence) on klassinen merkkijonoteorian ongelma, joka esitetään oppikirjallisuudessa usein esimerkkinä dynaamisen ohjelmoinnin (engl. dynamic programming) luvussa. Pisin yhteinen osajono on sellainen sarja symboleita, jotka löytyvät kahden syötteen tapauksessa samassa järjestyksessä molemmista syötteistä, riippumatta siitä onko niiden välissä jotain jakamattomia symboleita. Pisin yhteinen osajono on siis tiukempi vaatimus kuin yhteiset rivit, koska sen lisäksi, että rivit löytyvät molemmista tiedostoista, niiden pitää olla myös samassa järjestyksessä.

Pisin yhteinen osajono vastaa hyvin osakopion ongelmaan, koska se voidaan esittää suhteellisena arvona suhteessa vertailtavana olevan tiedostoparin yhteisten rivien määrään. Pisin yhteinen osajono ei voi olla pidempi kuin yhteiset rivit yhteensä. Pisin yhteinen osajono ei myöskään ole altis yksinkertaisille piilotusmuunnoksille kuten ylimääräisten rivien ripottelemiseen kopioidun koodin väliin.

Pisimmän yhteisen osajonon heikkoutena on ohjelman lohkojen järjestyksen muut-

taminen, jolloin pisin yhteinen osajono saadaan lyhenemään merkittävästi. Toinen huono puoli pisimmän yhteisen osajonon selvittämisessä on sen laskennallinen kompleksisuus. Dynaamisen ohjelmoinnin keinoilla päästään asymptoottiseen ajankulutusluokkaan $O(nm)$, jossa n ja m ovat vertailtavien tiedostojen pituudet. [2]

Cormen et.al [2] selittää seuraavasti dynaamisesta ohjelmoinnista: Dynaamisen ohjelmoinnin algoritmit muodostuvat kolmesta vaiheesta:

1. optimaalisen ratkaisun rakenteen hahmottelusta
2. rekursiivisesta optimaalisen ratkaisun määrittelystä
3. optimaalisen ratkaisun laskennasta alhaalta-ylös -mallilla.

Yksinkertaisuuden vuoksi määritellään $PYO(X_m, Y_n)$ tarkoittamaan jonojen X_m ja Y_n pisintä yhteistä osajonoa.

Optimaalisen ratkaisun rakenne voidaan hahmotella, jos optimaalisella ratkaisulla on optimaalinen osaratkaisu. Ongelmalla on optimaalinen osaratkaisu, jos ongelman optimaalinen ratkaisu sisältää optimaaliset ratkaisut ongelman osaongelmiin. Dynaamisessa ohjelmoinnissa lopullinen optimaalinen ratkaisu rakennetaan optimaalisten osaratkaisujen avulla, ja tämä on myös rekursiivisen määritelmän pohjana. Pisimmän yhteisen osajonon tapauksessa optimaalinen osaratkaisu voidaan määritellä seuraavasti: Jos syötteenä saadaan kaksi jonoa $X = \langle x_1, x_2, ..., x_m \rangle$ ja $Y = \langle y_1, y_2, ..., y_n \rangle$ ja $Z = \langle z_1, z_2, ..., z_k \rangle$ on mikä tahansa X:n ja Y:n pisin yhteinen osajono, niin

1. Jos $x_m = y_n$, niin $z_k = x_m = y_n$ ja $Z_{k-1} = PYO(X_{m-1}, Y_{n-1})$
2. Jos $x_m \neq y_n$, niin $z_k \neq x_m$ tarkoittaa että $Z = PYO(X_{m-1}, Y)$
3. Jos $x_m \neq y_n$, niin $z_k \neq y_n$ tarkoittaa että $Z = PYO(X, Y_{n-1})$

Rekursiivisen ratkaisu muodostetaan optimaalisen osaratkaisun avulla. Pisimmän yhteisen osajonon tapauksessa optimaalisen ratkaisun rakenne viittaa siihen, että riittää joko yhden tai kahden osaratkaisun tutkiminen: Jos $x_m = y_n$, niin lisäämällä $x_m = y_n$ jonojen X_{m-1} ja Y_{n-1} pisimpään yhteiseen osajonoon Z_{k-1} saadaan jonojen X ja Y pisin yhteinen osajono Z. Jos $x_m \neq y_n$, niin jonojen X ja Y pisin yhteinen osajono on pidempi jonojen X_{m-1} ja Y ja jonojen X ja Y_{n-1} pisimmistä yhteisistä osajonoista.

Rekursiivisen ratkaisun löytämiseksi määritellään $c[i, j]$:n olevan jonojen X_i ja Y_j pisimmän yhteisen osajonon pituus. Jos $i = 0$ tai $j = 0$, on pisimmän yhteisen osajonon pituus 0.

$$c[i,j] = \begin{cases} 0 & \text{jos } i = 0 \text{ tai } j = 0 \\ c[i-1, j-1] + 1 & \text{jos } i, j > 0 \text{ ja } x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{jos } i, j > 0 \text{ ja } x_i \neq y_j \end{cases} \tag{4.3}$$

Taulukko c voi kuitenkin kasvaa hyvin suureksi, koska lähdekooditiedostot voivat olla hyvinkin suuria, jopa kymmeniä tuhansia rivejä ja taulukon tilankäyttö on luokassa $O(nm)$, jossa n ja m ovat tiedostojen pituuksia. Kaavaa 4.3 tarkastelemalla huomataan kuitenkin, että taulukon arvo $c[i, j]$ riippuu vain edellisten rivien tai sarakkeiden arvosta, jolloin vain kaksi riviä tarvitaan kerrallaan, jolloin tilankäyttö supistuu luokkaan $O(n)$. Kaavan 4.3 toteutus on esitetty algoritmissa 4.2, jolle annetaan parametrina vertailtavat tiedostot A ja B. Tilankäytön optimointi näkyy algoritmin riveillä 4-5, joilla taulukko c on korvattu kahdella yksiulotteisella taulukolla, jotka kummatkin kuvaavat yhtä taulukon c riviä. Rivillä 18 rivit vaihdetaan keskenään ajankulutusluokassa $O(1)$, jolloin uudella silmukan kierroksella edellinen rivi on edellisen silmukan kierroksen aikana täytettävänä ollut rivi. Uuden silmukan kierroksen riviä ei tarvitse tyhjentää välissä, koska silmukka käydään aina loppuun asti, eikä vielä kirjoittamattomia taulukon alkioita käsitellä ennen kuin niihin on kirjoitettu uusi arvo.

```
 1: procedure LCS(A,B)
 2:     m ← length(A)
 3:     n ← length(B)
 4:     last ← Array[1..n + 1]
 5:     current ← Array[1..n + 1]
 6:     for i ← 1 to m + 1 do
 7:         for j ← 1 to n + 1 do
 8:             if i = 1 ∨ j = 1 then
 9:                 current[j] ← 0
10:             else
11:                 if A[i − 1] = B[j − 1] then
12:                     current[j] ← last[j − 1] + 1
13:                 else
14:                     current[j] ← MAX(current[j − 1], last[j])
15:                 end if
16:             end if
17:         end for
18:         SWAP(current, last)
19:     end for
20:     return last[n+1]
21: end procedure
```

Algoritmi 4.2 Algoritmi pisimmän yhteisen osajonon selvittämiseksi.

Kaavan 4.3 eri vaihtoehdot käsitellään algoritmissa 4.2 riveillä 5-10. Ensimmäinen tapaus käsitellään algoritmin riveillä 8-9, toinen riveillä 11-12 ja kolmas rivillä 14. Toteutusteknisistä syistä algoritmin 4.2 indeksit kulkevat yhden askeleen edellä yhtälöön 4.3 nähden.

4.4 Algoritmien yhdistäminen

Pisimmän yhteisen osajonon ja yhteisten rivien laskennan ajaminen yhdessä vastaa hyvin vertailun vaatimuksia eikä yhdistetyllä algoritmilla ole vakavia puutteita. Heikoin lenkki on suurten tiedostojen osakopiointi koodin järjestystä muuttamalla, jolloin yhdistetään molempien algoritmien heikoimmat puolet. Tätä voitaisiin torjua järjestämällä kutsutut funktiot kutsumisjärjestykseen, mutta tällöin ongelmaksi muodostuu kutsumisjärjestyksen selvittäminen sellaisissa tapauksissa, joissa ohjelma muodostuu useasta tiedostosta.

Ajankulutuksen kustannuksella voitaisiin ratkaista algoritmien heikoimmat puolet laskemalla kaikki pisimmät yhteiset osajonot, jotka eivät mene päällekkäin. Kaikkien pisimpien osajonojen yhteenlaskettu pituus on yhtä suuri kuin yhteisten rivien määrä, jolloin yhteisiä rivejä ei tarvitsisi laskea ollenkaan. Vertailun tulosta voitaisiin myös painottaa osajonojen pituuden perusteella. Kaikkien pisimpien osajonojen laskenta voitaisiin toteuttaa yksinkertaisesti laskemalla ja poistamalla vertailtavastasta tiedostoparista pisimpiä yhteisiä osajonoja niin kauan, kunnes osajonoja ei enää löydy lainkaan. Tämän algoritmin suoritusaika olisi kuitenkin kertaluokassa $O(n^3)$, joka olisi suurilla datamäärillä liian hidasta. Ronald I. Greenberg kuvaa julkaisussaan [4] algoritmin, jolla kaikkien pisimpien yhteisten osajonojen selvittäminen voidaan suorittaa kertaluokassa $O(nm)$.

Ohjelmakoodin järjestyksen muuttaminen on kuitenkin haastavaa ja rajoitettua muuten kuin kokonaisten aliohjelmien tapauksessa, jos ohjelman toiminnallisuuden täytyy pysyä samana. Ohjelmakoodilohkot ovat harvoin toisistaan niin riippumattomia, että ne voitaisiin sekoittaa täysin mielivaltaisesti, joka vaikeuttaa samankaltaisuuksien naamiointia.

Algoritmien heikkouksia voidaan myös välttää manuaalisen tulosten seulonnan avulla. Todellisten lukuarvojen seuraaminen auttaa löytämään tilanteita, joissa suhteellista arviointia on onnistuttu huijaamaan. Esimerkiksi muihin tuloksiin verrattuna pitkät yhteiset osajonot ovat epäilyttäviä, vaikka ne eivät vertailun kohteena olevien tiedostojen kokoon verrattuna olisikaan epätavallisia.

4.5 Tiedostotason algoritmit

Tähän mennessä olemme käsitelleet yksittäisen vertailualkioparin vertailua. Koska plagiaatteja etsiessä vertailualkioparien valintaa ei voi ennustaa etukäteen, jokainen vertailualkio on vertailtava jokaista toista vertailualkiota vastaan. Ohjelmakoodi voidaan jakaa useampaan tiedostoon, ja eräs päätös onkin, onko vertailualkiona kokonainen ohjelma vai tiedosto. Ohjelmakoodi voidaan jakaa useaan eri tiedostoon, ja algoritmien toteutuksen ja valinnan kannalta on ratkaisevaa, valitaanko vertailualkioksi kokonainen ohjelma vai tiedosto. Jos yksiköksi valitaan kokonainen ohjelma, asetetaan vertailutyökalulle tiukempia vaatimuksia, koska tällöin esimerkiksi neliöllisten algoritmien suoritusaika kasvaa nopeasti ja osittaisten kopiointien löytäminen on vaikeampaa. Toisaalta kokonaisen ohjelman valinta auttaa löytämään tapauksia, joissa plagioija on jakanut alun perin yhdessä tiedostossa olleen ohjelmakoodin useampaan tiedostoon.

Jos tiedostoja käytetään vertailualkiona, muodostaa jokainen tiedosto algoritmien kannalta oman kokonaisuutensa. Tämä auttaa osittaisten kopiointien löytämisessä erityisesti sen vuoksi, että samankaltaisuudet ilmoitetaan usein suhteellisena lukuna, jolloin suurella ohjelmakoodimäärällä voidaan peittää osittaisia kopiointeja. Riippumatta valitusta yksiköstä, alkioiden vertailu on aina ajankulutusluokassa $O(n^2)$, jossa n on vertailualkioiden lukumäärä. Samaan ohjelmaan kuuluvat tiedostot voidaan plagiaatteja etsiessä jättää vertailematta keskenään ajan säästämiseksi, koska itsensä toistaminen ei ole kiellettyä. Tässä työssä valittiin vertailualkioksi yksi tiedosto.

```
1:  procedure CompareFiles(A)
2:      B ← {a | ∀a ∈ A}
3:      for all s1 ∈ A do
4:          for all s2 ∈ B do
5:              if s1 ≠ s2 then
6:                  for all f1 ∈ s1 do
7:                      for all f2 ∈ s2 do
8:                          compare(f1, f2)
9:                      end for
10:                 end for
11:             end if
12:         end for
13:         B ← B \ {s1}
14:     end for
15: end procedure
```

Algoritmi 4.3 Algoritmi tiedostojen vertailuun

Tiedostojen vertailu voidaan suorittaa algoritmin 4.3 mukaisesti. Joukko A sisältää

kaikki vertailujoukkoon kuuluvat ohjelmat. Jokainen joukon A alkio sisältää joko yhden tai useamman tiedoston joukon. Jokaista ohjelmaa verrataan kaikkiin muihin ohjelmiin paitsi itseensä. Vertailua ei tarvitse tehdä kuin yhteen suuntaan, joten jos palautukset s_1 ja s_2 on jo vertailtu, ei vertailua tarvitse suorittaa uudestaan.

4.6 Samankaltaisuuden mittarit

Mittareiden avulla pyritään mittaamaan parien samankaltaisuutta niin, että tulosjoukosta voidaan nostaa esiin eniten samankaltaiset parit. Mittareiden tarkoituksena on lisäksi visualisoida varmistajalle samankaltaisuutta. Samankaltaisuutta voidaan mitata yhdellä tai useammalla mittarilla, mutta kuten vertailualgoritmien tapauksessa, myös mittareilla on hyvät ja huonot puolet. Yhdellä mittarilla mitattuja tuloksia on helppo tulkita, mutta mittarista voi muodostua niin monimutkainen, että sen muodostumisperiaatteita on vaikea tunnistaa. Toisaalta useammalla mittarilla tulokset voivat olla jopa ristiriitaisia, riippuen niiden painotuksista. Mittareita tulkittaessa epätavallisesti muista erottuvat arvot ovat aina syy tarkastella lähdekoodia käsin.

$$S(f_1, f_2) = \frac{yht(f_1, f_2)}{\sum_{\langle x,i\rangle \in f_1} i} * 100\%, \sum_{\langle x,i\rangle \in f_1} i \neq 0 \tag{4.4}$$

Yhteisten rivien algoritmilla tiedoston f_1 samankaltaisuutta tiedostoon f_2 nähden voidaan mitata yhtälöllä 4.4, jossa yhtälön 4.2 avulla lasketaan tiedostojen yhteisten rivien määrän suhde tiedoston f_1 kaikkien rivien määrään. Yhtälön 4.4 tulos siis kertoo, kuinka suuri osa tiedostosta f_1 löytyy myös tiedostosta f_2. Samankaltaisuus lasketaan molempiin suuntiin, koska tiedostojen rivimäärät ja jaettujen rivien suhteellinen osuus voivat vaihdella huomattavasti. Esimerkiksi jos $S(f_1, f_2) = 100\%$ ja $S(f_2, f_1) = 50\%$, tiedetään, että tiedoston f_1 sisältö löytyy kokonaisuudessaan tiedostosta f_2, mutta vain puolet tiedoston f_2 sisällöstä löytyy myös tiedostosta f_1.

$$S_{pyo}(f_1, f_2) = \frac{c[length(f_1), length(f_2)]}{yht(f_1, f_2)} * 100\%, yht(f_1, f_2) \neq 0 \tag{4.5}$$

Samankaltaisuuden mittaaminen pisimmän yhteisen osajonon avulla tehdään yhtälön 4.3 avulla siten, että jonot X_m ja Y_n muodostetaan tiedostojen f_1 ja f_2 lähdekoodiriveistä siinä järjestyksessä, missä ne esiintyvät alkuperäisissä lähdekooditiedostoissa. Tiedostojen f_1 ja f_2 samankaltaisuus pisimmällä yhteisellä osajonolla mitattuna saadaan yhtälöllä 4.5. Pisimmän yhteisen osajonon avulla samankaltaisuus

lasketaan vain kerran, koska yhtälö 4.5 antaa tuloksen suhteessa tiedostojen f_1 ja f_2 yhteisiin riveihin, jolloin tulos olisi sama kumpaankin suuntaan laskettuna.

Mitattujen samankaltaisuusarvojen lisäksi yhtälöissä käytetyt numeeriset arvot kannattaa näyttää vertailun suorittajalle, jolloin suhteellisuuden luonne saadaan paremmin tuotua esiin. Numeerisilla arvoilla vertailun suorittaja voi nopeasti seuloa pois hyvin pienet tiedostot ja hyvin vähän samoja rivejä samassa järjestyksessä sisältävät tiedostot.

5. VERTAILUTYÖKALU

Työssä suunniteltiin vertailutyökalu Plakki 1.0 uudelleen. Ohjelman tietorakenteet rakennettiin uudelleen, vertailualgoritmit vaihdettiin tehokkaampiin ja käsitekieltä korjattiin. Uudelleensuunnittelussa painotettiin erityisesti tehokkuutta, joka oli suuri ongelma ohjelmaa käytettäessä.

Tässä luvussa kuvataan lyhyesti Plakki 1.0:n toiminta sekä esitellään uuden version 2.1 toiminta ja tehdyt muutokset.

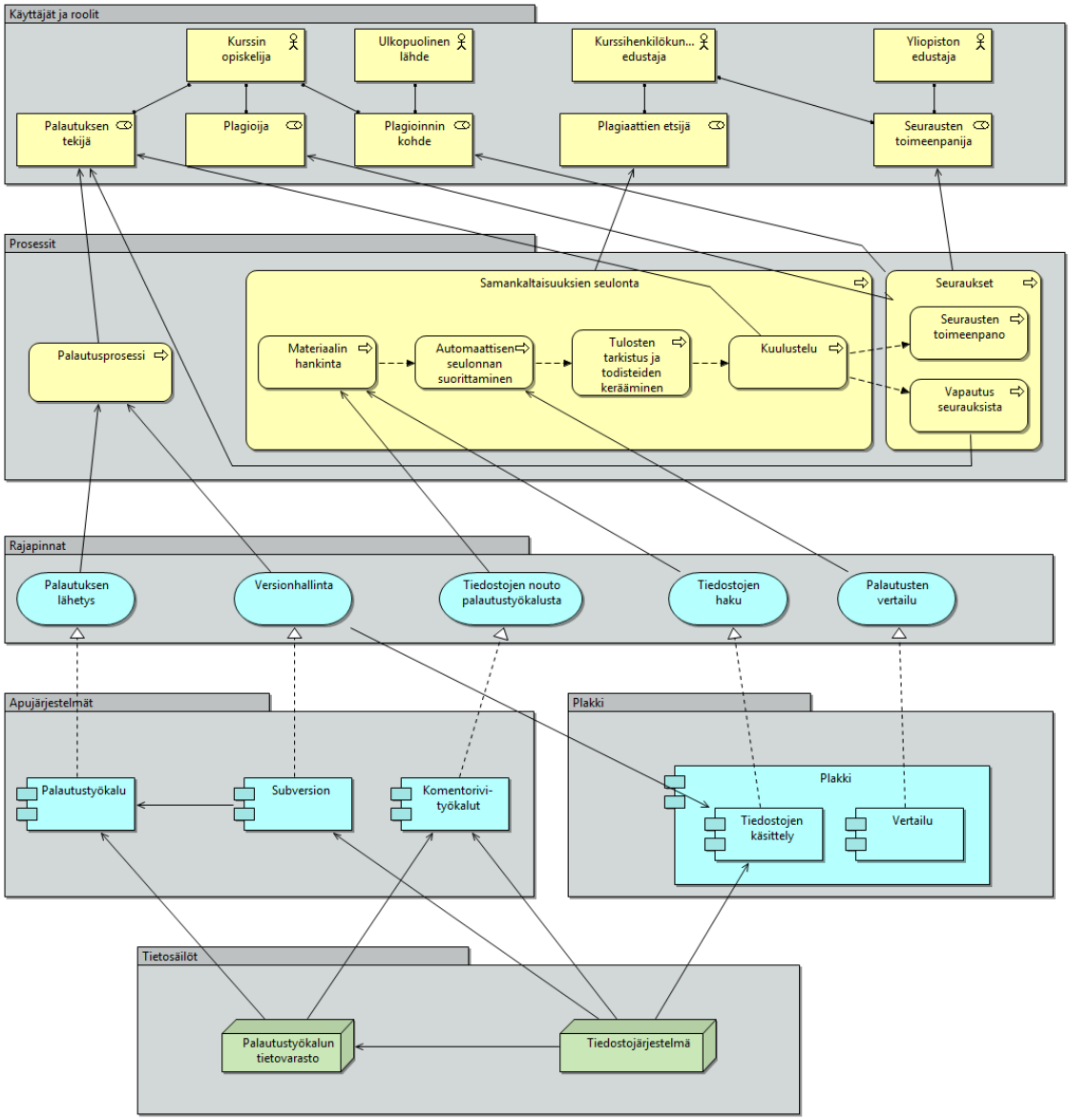
5.1 Lähtötilanne

Plakin versiosta 1.0 ei ole tuotettu teknistä dokumentaatiota, mutta lyhyt kuvaus järjestelmästä on kirjattu TTY:n laitosraporttiin [11]. Plakki 1.0 on toteutettu kokonaan Python-kielellä ja se toimii komentoriviltä käytettävänä ohjelmana Unix-käyttöjärjestelmässä. Plakki 1.0 perustui omaan ohjelmointikieliriippumattomaan käsitekieleen kuten on kuvattu luvussa 3.2.3 ja tiedostojen vertailussa käytettiin yhteisten rivien algoritmia. Plakki 1.0 kykeni vertailemaan vain C/C++-kielillä kirjoitettuja ohjelmia. Plakissa 1.0 kaikki tiedostot kirjoitettiin takaisin kiintolevylle käsitekielisessä muodossa ja tämän jälkeen jokainen tiedostopari syötettiin Comm-ohjelmalle, joka tulostaa parametrinaan saamistaan tiedostoista sellaiset rivit, jotka löytyvät molemmista tiedostoista. Commin tuloste luettiin vielä takaisin tietokoneen muistiin tulosjoukon muodostamista varten. Lopuksi käsitekieliset tiedostot poistettiin kiintolevyltä.

Kuten TTY:n laitosraportin kirjoittajat toteavat, Plakin 1.0 tehokkuudessa oli parantamisen varaa. Käytännössä yhden ohjelmointikurssin töiden vertailuun kului seinäkelloaikaa useita tunteja. Vaikka Plakkia tyypillisesti käytetään yhden kurssin osalta vain kerran kurssin suorituksen aikana, aiheutti epätehokkuus tarpeetonta ajankulua tapauksissa, joissa vertailun suorittaja halusi suorittaa plagiaattien etsinnän samalle vertailujoukolle erilaisilla asetuksilla.

5.2 Kokonaisarkkitehtuuri ja käyttötarkoitus

Plakkia 2.1 käytetään TTY:n ohjelmistotekniikan laitoksen perusohjelmointikurssien yhteydessä kurssihenkilökunnan aputyökaluna ja sen käyttötarkoituksena on seuloa suuresta määrästä lähdekooditiedostoja toisiaan muistuttavat tiedostot, jotta mahdolliseen plagiointiin voidaan puuttua. Plakkia 2.1 voidaan käyttää missä tahansa ohjelmien lähdekooditiedostojen samanlaisuuden vertailua vaatimassa operaatiossa. Kuvassa 5.1 on kuvattu plagiaattien etsinnän kokonaisarkkitehtuuri, jota on sovellettu joillakin TTY:n ohjelmistotekniikan laitoksen kursseilla. Kokonaisarkkitehtuurin esittämiseen on käytetty Archimate-kuvauskieltä [5].



Kuva 5.1: Kokonaisnäkymä samankaltaisuuksien etsinnän prosessista

Plagiaattienetsinnän perusajatuksena on luvussa 3.1 kuvattu prosessi. Toimintamal-

lia esitetään otettavaksi käyttöön kaikille ohjelmointikursseille, jolloin plagiaattien etsintä olisi systemaattista. Aiemmin on ollut tapana, että jokainen kurssin vastuuhenkilö päättää itse, miten plagiointitapauksia etsitään.

Kurssin opiskelijat palauttavat kurssia suorittaessaan työnsä automaattiseen palautusjärjestelmään tai suoraan kurssihenkilökunnan edustajalle palautteen antoa varten. Joillakin kursseilla palautuksessa ei toimiteta koko ohjelman lähdekoodia, vaan linkki kirjoitussuojattuun lähdekoodin versioon versionhallinnassa. Kurssin päätyttyä kurssin vastuuhenkilön nimeämä vertailija kerää ensin kaikki kurssille tehdyt palautukset palautusjärjestelmästä käsin, tai antaa Plakin 2.1 hakea palautukset tiedostojärjestelmästä tai suoraan versionhallinnasta. Tämän jälkeen vertailun suorittaja käyttää Plakkia 2.1 ja ryhtyy etsimään käsin todisteita plagioinnista Plakin 2.1 antaman tuloslistan perusteella. Jos todisteita plagioinnista löytyy, kutsutaan osapuolet kuultaviksi ja tehdään päätös, joka voi olla langettava tai vapauttava.

5.3 Yleiskuvaus järjestelmästä

Plakin 2.1 perustoimintaperiaate on sama kuin versiossa 1.0. Plakki 2.1 koostuu kolmesta eri komponentista: vertailtavien tiedostojen tunnistamisesta, niiden muunnoksesta käsitekieleksi ja varsinaisesta tiedostojen vertailusta. Tiedostojen tunnistamisen ja käsitekielen muunnoksen toteutuskielenä on Python ja vertailuosa on toteutettu C++-ohjelmointikielellä tehokkuussyistä. Ohjelma voidaan suorittaa myös täysin alustariippumattomasti vivulla -s, jolloin myös vertailu suoritetaan Python-kielellä. Alustariippumaton ajo aiheuttaa kuitenkin huomattavan ajankulutuksen suuria lähdekoodimääriä vertailtaessa, joten pisimmän yhteisen osajonon algoritmi on kytketty alustariippumattomassa ajossa pois päältä.

Vertailtavat tiedostot haetaan annetusta hakemistohierarkiasta siten, että jokaiselle tiedostolle määritellään omistaja, jos mahdollista. Omistajan tiedostoja ei tarvitse vertailla keskenään. Jos omistajaa ei voida päätellä hakemistopolusta, ei tiedostolla ole omistajaa, jolloin se vertaillaan kaikkia tiedostoja vastaan. Plakki 2.1 kykenee hakemaan tiedostot myös Subversion -versionhallinnasta. Subversion-ominaisuus vaatii kuitenkin ylimääräistä muistia työkopioita varten.

5.4 Tietorakenteet ja algoritmit

Plakki 2.1 käsittelee tiedostoja omassa käsitekielisessä esitysmuodossaan kuten vanhassakin versiossa, mutta uuden version osalta käsitekieltä laajennettiin muutamien

puutteiden osalta. Plakissa 2.1 tietorakenteiden avainajatus on mallintaa vertailtavan käsitekielisen tiedoston sisältö tiivistetyssä muodossa. Ajatuksena on, että jokainen rivi saa ajokertakohtaisesti lasketun täydellisen hajautusarvon, jolloin jokainen rivi voidaan esittää yhdellä kokonaisluvulla. Käytännössä täydellisen hajautusarvon antaminen ei ole kaikissa tapauksissa mahdollista tietokoneiden kokonaisluvun arvoalueen rajallisuuden vuoksi, mutta ratkaisu toimii niin kauan kuin vertailuun mukaan otettavien ohjelmakoodirivien määrä mahtuu ohjelmaa ajavan tietokoneen käyttämän kokonaisluvun arvoalueeseen. Jokaisen tiedoston samat rivit lasketaan ja näistä muodostetaan kohdassa 4.2 kuvatulla tavalla järjestetty monijoukko rivitunniste - esiintymismäärä -kokonaislukupareja kokonaisten tekstirivien sijasta, joiden määrä on aina pienempi tai yhtä suuri kuin alkuperäisten tekstirivien määrä. Tiivistetty muoto ei hävitä vertailun kannalta mitään olennaista, mutta pienentää ohjelman muistinkulutusta ja nopeuttaa vertailua huomattavasti. Yhteisten rivien algoritmi käsittelee monijoukkoon tiivistettyä muotoa.

Plakki 2.1 käyttää vertailuun myös pisimmän yhteisen osajonon algoritmia, joka on kuvattu kohdassa 4.3. Pisimmän yhteisen osajonon algoritmi tarvitsee rivien alkuperäisen järjestyksen, joten tiedostoista täytyy pitää monijoukkoesityksen lisäksi myös alkuperäisessä järjestyksessä oleva, yksikäsitteisiksi kokonaislukutunnisteiksi muutettu muoto. Tiedoston sisällön pitäminen kahdessa eri muodossa kuluttaa muistia, mutta muistinkulutus on silti yleensä pienempi kuin alkuperäisten merkkijonojen säilyttäminen.

5.5 Esitystapa

Oletustapauksessa Plakki 2.1 laskee kaikille vertailujoukossa mukana oleville tiedostoille samankaltaisuusarvot sivulla 26 määriteltyjen yhtälöiden 4.4 ja 4.5 avulla. Tiedostoparit järjestetään listaan, jossa suuret samankaltaisuusarvot esiintyvät ensin ja pienet lopuksi. Yhden tiedostoparin samankaltaisuutta esittävä rivi jakautuu kolmeen kaksoispisteellä erotettuun osaan:

- Yhteisten rivien samankaltaisuus esitetään molempien tiedostojen suhteen yhtälön 4.4 avulla muodossa $s(f_1, f_2)/s(f_2, f_1)$
- Pisimmän yhteisen osajonon samankaltaisuus esitetään yhtälön 4.5 avulla muodossa $s_{pyo}(f_1, f_2)(c[length(f_1), length(f_2)], yht(f_1, f_2))$
- Tiedostojen yksilöintitiedot ja numeeriset rivimäärät esitetään lopuksi siten, että tiedoston nimen jälkeen annetaan sulkeiden sisällä tiedoston rivimäärä.

Esimerkkilistaus tuloksista nähdään kuvassa 5.2

```
100% / 100%  :  91% ( 34,  37 ): student_116_salaus.cc ( 37 ) student_27_salaus.cc ( 37 )
 95% /  93%  : 100% ( 44,  44 ): staff_3_salaus1.cc ( 46 ) staff_root_salaus.cc ( 47 )
 90% /  85%  : 100% ( 40,  40 ): staff_2_salaushelppo.cc ( 44 ) staff_root_salaus.cc ( 47 )
 88% /  84%  : 100% ( 39,  39 ): staff_2_salaushelppo.cc ( 44 ) staff_3_salaus1.cc ( 46 )
 91% /  86%  :  90% ( 29,  32 ): student_2_salaus.cc ( 35 ) student_61_salaus.cc ( 37 )
 81% /  90%  :  93% ( 28,  30 ): student_116_salaus.cc ( 37 ) student_119_salaus.cc ( 33 )
 90% /  83%  :  90% ( 27,  30 ): student_51_salaus.cc ( 33 ) student_95_salaus.cc ( 36 )
 91% /  91%  :  81% ( 26,  32 ): student_2_salaus.cc ( 35 ) student_54_salaus.cc ( 35 )
```

Kuva 5.2: Plakin 2.1 tulosten esitystapa

Oletustapauksessa lista järjestetään yhteisten rivien ja pisimmän yhteisen osajonon suhteellisten arvojen yhdistelmänä.

5.6 Apuominaisuudet

Plakkiin 2.1 on liitetty mahdollisuus seuloa tiedostoja yksinkertaisella kompleksisuusmitalla. Oletuksena seulontaa ei käytetä, mutta haluttu kompleksisuustaso voidaan asettaa. Jokaisesta tiedostosta selvitetään kontrollirakenteiden määrä, ja jos tiedoston kompleksisuusaste ylittää annetun rajan, tiedosto otetaan mukaan vertailuun. Kompleksisuusmitalla voidaan poistaa hyvin yksinkertaisia ja yleiskäyttöisiä tiedostoja, jotka oletetusti saavat keskenään korkeita samankaltaisuusarvoja.

Plakille 2.1 voi antaa tiedostoja tai hakemistoja mustaan listaan, jolloin kyseisiä tiedostoja tai hakemistoista löytyviä tiedostoja ei oteta mukaan vertailuun. Tämä selkeyttää tuloksien tarkastelua huomattavasti tilanteissa, joissa kurssihenkilökunta antaa osan toteutettavasta järjestelmästä valmiina.

Plakille 2.1 voidaan antaa parametrina ehtoja tulosjoukkoon päätymiselle. Pisimmälle yhteiselle osajonolle voidaan antaa minimipituus sekä kaikkia suhteellisten mittareiden rajoja voidaan säätää. Plakin 2.1 antaman tulosjoukon maksimikokoa voidaan myös säätää. Käsitekielisen rivin pituus voidaan myös asettaa vakioksi, mutta oletuksena oleva dynaaminen rivien pituuden päättely on suositeltava valinta, kuten on selitetty kohdassa 4.1.

5.7 Muunneltavuus

Työkalun suunnittelussa pyrittiin säilyttämään uusien ohjelmointikielten lisäämisen helppous, jonka vuoksi käsitekieliset muunnokset on toteutettu pääosin säännöllisillä lausekkeilla. Säännöllisillä lausekkeilla ei kuitenkaan pysty helposti suorittamaan tiettyjä muunnoksia, joten sääntöjoukkoihin voi lisätä myös Python-kielellä kirjoitettuja funktioita. Uuden kielen lisääminen on yksinkertaista ohjelmarakenteen tasolla,

koska lisääminen vaatii ainoastaan muunnossääntöjen ja niitä vastaavan tiedostopäätteen lisäämisen ohjelmaan. Suuri osa muunnossäännöistä käy nykyisellään eri ohjelmointikieliin, yleensä ainoastaan varattujen sanojen ja operaattorien listaa voi joutua päivittämään. Muunnoslistaa luotaessa on suotavaa kiinnittää muunnosten järjestykseen huomiota. Plakki 2.1 tukee seuraavia kieliä: C, C++, Java, Python ja JavaScript.

Mittareiden ja järjestyssääntöjen puolesta Plakin 2.1 muunneltavuus on heikkoa tai keskitasoa, ja vaatii syvää tutustumista Plakin 2.1 tietorakenteisiin ja algoritmeihin. Uuden algoritmin lisääminen vaatii itse algoritmin kirjoittamisen lisäksi muutoksia tulosten tallennuslogiikkaan ja järjestysfunktioihin. Uusien algoritmien lisääminen olisi teknisesti helppo toteuttaa, mutta algoritmin tulosten tulkitseminen ja järjestysfunktioiden muotoilu on ongelmallista, koska näissä pitäisi ottaa huomioon myös muiden algoritmien antamat tulokset. Yksi ratkaisu olisi muuttaa järjestystoiminnallisuus sellaiseksi, että järjestyksen määräisi yksi arvo, joka lasketaan kaikkien vertailuun mukaan otettavien algoritmien tulosten funktiona siten, että eri algoritmien tuloksia voidaan painottaa painokertoimella. Tämä ratkaisu kuitenkin poistaa kokonaan toisistaan riippumattomien metriikoiden kyvyn erottaa poikkeuksellisen korkeita arvoja.

Toinen ratkaisu olisi muuttaa Plakin 2.1 toimintaprosessia siten, että yhdellä vertailukerralla vertaillaan lähdekoodeja vain yhden algoritmin avulla, joka valitaan jokaiselle suorituskerralle satunnaisesti Plakkiin 2.1 kirjoitetusta vertailualgoritmijoukosta. Tämä ratkaisu vähentäisi plagioijan mahdollisuuksia hyväksikäyttää tietyn vertailualgoritmin heikkoutta, mutta toisaalta se vaikeuttaisi Plakin 2.1 käyttöä tulosten tulkinnan osalta.

5.8 Jatkokehitysideoita

Plakissa 2.1 on vielä paljon parannettavaa, suurimpana heikkoutena Plakki 2.1:llä ei ole graafista käyttöliittymää. Graafinen käyttöliittymä helpottaisi tarkistajan työtä huomattavasti erityisesti silloin, jos käyttöliittymässä näytettäisiin vertailtujen tiedostojen listaus siten, että vastaavat rivit olisivat korostettu muusta ohjelmakoodista erottuvalla värillä. Plakin 2.1 vertailualgoritmeissa on nykyisessä versiossa vielä heikkouksia, joita hyödyntämällä plagioija voi välttää kiinnijäämisen.

Plagiointitapausten tunnistuksen osalta työkalun suurimpia puutteita on kyvyttömyys havaita muualla teetettyä ohjelmakoodia. Ongelmaa voitaisiin ratkaista luomalla jokaiselle opiskelijalle oma ohjelmointikäsiala analysoimalla lähdekoodia ja sen kehittymistä harjoitustöissä. Tyypillisesti ohjelmoijat käyttävät vain pientä osaa

kielestä hyödykseen. Muita havaittavia ominaisuuksia voisi olla muun muassa funktioihin jakaminen, lauseiden monimutkaisuus, muuttujien käyttö. Edellä mainittua työkalua voitaisiin käyttää myös muihin tarkoitukseen kuin plagiaattien tunnistamiseen. Ohjelmointikäsiala voisi kertoa ohjelmoijan kypsyydestä esimerkiksi muuttujien käytön järjestelmällisyyden analyysin kautta.

6. VERTAILU

Plakin 2.1 todellista suoriutumista plagiaattien löytämisessä on vaikea arvioida pelkästään teoreettiselta pohjalta, joten Plakin 2.1 suoriutumista verrattiin kahteen tunnettuun plagiaatinpaljastimeen siten, että kaikilla järjestelmillä vertailtiin samat vertailujoukot. Vertailun tarkoitus oli selvittää pääosin algoritmien kyky erotella plagiaatteja, eikä huomioon otettu juurikaan käytettävyyteen liittyviä asioita. Tässä luvussa esitellään vertailussa käytetyt muut järjestelmät sekä kuvataan vertailuun mukaan otetut vertailujoukot. Lopuksi esitellään vertailun tulokset.

6.1 Muut järjestelmät

Monilla yliopistoilla on käytössään jonkinlainen vertailutyökalu tai menetelmä, jolla samankaltaisuuksia etsitään. Usein työkalut ovat yliopistojen itse kehittämiä, mutta muutamia järjestelmiä on käytössä useammissa paikoissa. Tähän vertailuun valittiin Stanfordin yliopistossa kehitetty Moss [10] ja Karlsrühen yliopistossa kehitetty JPlag [9]. Valinta perustellaan sillä, että molemmista järjestelmistä on tehty julkaisu, ne eivät vaadi asennuksia, ne ovat ilmaisia ja molempiin löytyy lukuisia viittauksia Internetin hakukoneilla.

6.1.1 JPlag

JPlag perustuu samantapaiseen lähdekoodin muuntamiseen kuin Plakki 2.1. JPlagissa lähdekoodia ei käsitellä samalla tavoin eri ohjelmointikielissä, vaan apuna on kielestä riippuen kääntäjä tai skanneri. Ohjelmakoodista luodaan sarja ohjelmointikielestä muodostettuja merkkejä (engl. token), joiden on tarkoitus kuvata ohjelman rakennetta mahdollisimman hyvin. JPlag käsittelee ohjelmia kokonaisuuksina ja kahden ohjelman samankaltaisuutta etsitään ahneella merkkijonolaatikoinnilla (engl. Greedy String Tiling).

Matches sorted by average similarity (What is this?):

student_27	->	student_116 (100.0%)	student_89 (71.4%)	student_54 (69.8%)	student_121 (54.6%)	student_119 (50.0%)
student_11	->	student_20 (77.6%)	student_109 (77.2%)	student_103 (61.6%)	student_40 (57.1%)	student_21 (52.2%)
staff_root	->	staff_3 (74.4%)	staff_4 (55.5%)	staff_2 (52.1%)		
student_109	->	student_20 (72.9%)	student_103 (64.5%)	student_40 (55.9%)	student_21 (52.5%)	
student_116	->	student_89 (71.4%)	student_54 (69.8%)	student_121 (54.6%)	student_119 (50.0%)	student_2 (48.4%)

Kuva 6.1: JPlagin tulosten esitystapa

JPlag esittää vertailun tulokset kaksiulotteisesti kuvan 6.1 mukaisesti. Kaksiulotteinen esitysmuoto auttaa löytämään niin kutsuttuja kopiointirinkejä, joissa useampi opiskelija on plagioinut samasta lähteestä. JPlag on toteutettu Java-kielellä.

6.1.2 Moss

Moss perustuu tekijöidensä kehittämään paikalliseen sormenjälkialgoritmiin Seulonta (engl. Winnowing). Ajatus on että vertailtavan teksti jaetaan k:n merkin mittaisiin uniikkeihin alimerkkijonoihin, joille lasketaan tiivistearvo. Lasketuista tiivisteistä muodostetaan sekvenssi, jota etsitään Karp-Rabinin merkkijonoalgoritmilla muista tiedostoista.

```
student_103/ (64%)   student_20/ (66%)    99
student_103/ (61%)   student_109/ (59%)   96
student_109/ (57%)   student_20/ (60%)    79
student_21/ (28%)    student_40/ (31%)    42
student_11/ (30%)    student_20/ (30%)    47
student_22/ (45%)    student_64/ (35%)    26
staff_2/ (67%)   staff_root/ (62%)    43
student_109/ (23%)   student_11/ (25%)    37
```

Kuva 6.2: Mossin tulosten esitystapa

Moss esittää vertailun tulokset Plakin 2.1 tapaan järjestettynä listana kuvan 6.2 tapaan. Mossin toiminnasta ja järjestysmenetelmästä ei ole julkista dokumentaatiota,

mutta esimerkkilistauksen perusteella voidaan tulkita sen perustuvan useampaan erilaiseen muuttujaan.

6.2 Testijoukot

Testijoukkoina vertailussa on käytetty neljää erilaista joukkoa, joista kolme ensimmäistä koostuu opiskelijoiden todellisista C++-kielisiä ohjelmakoodipalautuksia TTY:n laajan ohjelmoinnin kursseilta. Lisäksi kahteen joukkoon on osa TTY:n ohjelmistotekniikan laitoksen henkilökunta toimittanut kumpaankin kaksi eritasoista plagiaattia, joista toinen pyrkii käyttämään heikon opiskelijan taidoille tyypillisiä muunnoksia ja toinen taitavan, mutta heikosti motivoituneen opiskelijan oletettuja taitoja. Neljäntenä tapauksena käsitellään Plakin 2.1 tehokkuutta ja kykyä löytää samankaltaisuuksia suurella testijoukolla. Valitettavasti erittäin suuren testijoukon vertaileminen Mossiin ja JPlagiin ei onnistunut teknisten rajoitusten vuoksi.

Testijoukko A sisältää 126 opiskelijoiden tekemää palautusta ja 7 henkilökunnan palautusta, joista yksi on plagioitavaksi annettu ratkaisu. Testijoukko A:n tehtävänantona oli toteuttaa sekoitettu Caesar-salaus, jossa syötteenä annetaan ensin salausavaimella korvaava merkki jokaiselle englannin kielen aakkoselle. Tämän jälkeen käyttäjältä kysytään salattavia sanoja, ja ohjelma tulostaa sanat salattuina näytölle. Tehtävän tarkoituksena oli harjoitella silmukoiden ja taulukoiden käyttöä, ja lopputuloksena on suhteellisen lyhyt, mutta aloittelijoille monimutkainen ohjelma. Tehtävän heikkoutena on herkkyys virheille, jos ohjelmaa ei toteuta juuri oikealla tavalla, mikä saattaa näkyä korkeammissa samankaltaisuusarvoissa. Tehtävä on ollut käytössä ohjelmoinnin ensimmäisellä kurssilla kolmantena harjoitustyönä. Testijoukossa A on kuusi teetettyä plagiaattia ja yksi todettu opiskelijoiden välinen plagiointitapaus.

Testijoukko B sisältää 127 opiskelijoiden palautusta ja 5 henkilökunnan tekemää palautusta, joista yksi on plagioitavaksi annettu ratkaisu. Testijoukko B:n tehtävänantona oli toteuttaa yksinkertainen komentorivipohjainen komentotulkki, jossa useammalla eri komennolla suoritetaan sama toiminto. Komentoja pitää voida lyhentää siten, että ohjelma ymmärtää myös kaikki alimerkkijonot, jotka voidaan yksikäsitteisesti yhdistää yhden komennon etuliitteeksi. Tehtävän tarkoituksena oli harjoitella merkkijonojen käsittelyä, luettelotyypin **enum** käyttöä ja laajennettavuutta. Lopputuloksena oletetaan suhteellisen lyhyttä ja suoraviivaista ohjelmaa, mutta myös huomattavasti työläämpiäkin ratkaisuja esiintyy. Tehtävä on ollut ohjelmoinnin toisella kurssilla kolmantena harjoitustyönä. Testijoukossa B on neljä teetettyä plagiaattia.

Testijoukkojen A ja B tehtävänannot ovat kuuluneet kurssien laaja ohjelmointi 1 ja 2 ohjelmaan vuosina 2009-2012 ja tätä ennen samat tehtävänannot olivat kursseilla ohjelmointi 1e ja 2e. Uudelleenkäyttö useampana vuonna lisää kopiointitapauksia, koska aiempien lukuvuosien opiskelijat saattavat auttaa nuorempiaan. Tehtävät A ja B ovat olleet niin kutsuttuja palautustehtäviä, pienehköjä ohjelmia, jotka palautetaan vain automaattiselle tarkastustyökalulle. Tyypillisesti suurin osa kurssien ohjelmointi 1e ja laaja ohjelmointi 1 kopiointitapauksista ovat olleet palautustehtävissä.

Testijoukko C sisältää 103 opiskelijoiden tekemää palautusta. Testijoukko C:n tehtävänä oli toteuttaa laajan ohjelmoinnin kurssin suurempi harjoitustyö, TTY:n henkilökunnan yksinkertaistama merkkipohjainen versio vuonna 1988 julkaistusta pelistä Ataxx. Tehtävän tuloksena on jo joukkoihin A ja B nähden huomattavasti monimutkaisempi ja laajempi ohjelma. Testijoukko C on muodostettu eri lukuvuoden palautuksista kuin A ja B siksi, että näin jokaiseen testijoukkoon saatiin vähintään yksi todellinen plagiointitapaus, jonka asianomaiset ovat myöntäneet plagioinnin. Testijoukko C on kurssin päättävä harjoitustyö, joka on laajuudeltaan huomattavasti palautustehtäviä suurempi. Harjoitustyössä opiskelijat tekevät myös kirjallisen suunnitelman, josta kurssihenkilökunnan edustaja antaa palautetta ja tarvittaessa ohjaa opiskelijaa eteenpäin. Harjoitustöissä on harvoin ollut kopiointitapauksia, mikä saattaa selittyä opiskelijan paremmalla ohjauksella. Testijoukossa C on yksi todettu opiskelijoiden välinen plagiaatti.

Testijoukko D sisältää Android -käyttöjärjestelmän version 3.1.4.1.5.9.2.6.5 ja Java SE 6u23 b05 lähdekoodien vertailun. Testijoukko D suoritettiin, kun Oracle haastoi Googlen tekijänoikeuksista oikeuteen väitteenään, että Google olisi käyttänyt luvatta osia Javan lähdekoodeista Android -käyttöjärjestelmän toteutuksessa [1]. Testijoukko D on monella tapaa erilainen kuin muut testijoukot, koska se sisältää miljoonia rivejä ohjelmakoodia, mutta vain kaksi eri lähdekoodin omistajaa. Vertailuun otettiin mukaan Java-kielellä kirjoitetut tiedostot, mutta myös muilla kielillä kirjoitettuja osia on erityisesti Androidissa paljon. Tiedostoja Androidissa on 19272 ja ohjelmakoodirivejä 4 051 410, Java SE:ssa vastaavat luvut ovat 11318 ja 3 276 751. Koska molemmat järjestelmät sisältävät erittäin paljon hyvin pieniä tiedostoja, joissa on pääasiassa poikkeus- ja rajapintaluokkien määrittelyitä, otettiin joukkoon vain sellaiset tiedostot, joiden kompleksisuusaste on vähintään viisi. Kompleksisuusseulonnan jälkeen Androidin tiedostoista päätyi testijoukkoon 7112 ja Java SE:stä 5560.

6.3 Tavoitteet

Vertailun tavoitteena on selvittää, pystyykö Plakki 2.1 havaitsemaan plagiaatteja yhtä hyvin kuin tunnetut plagiaatintarkistimet JPlag ja Moss. JPlagia ja Mossia on vertailtu useaan otteeseen, joten tässä vertailussa keskitytään vain Plakin 2.1 toimintaan JPlagin ja Mossin suhteen. Tarkoituksena on selvittää, löytävätkö plagiaatintarkistimet kaikki todelliset plagiaatit niin, että ne päätyvät epäiltyjen listalle. Koska testijoukot A ja B ovat hyvin yksinkertaisia, on oletettavaa, että tarkastustyökalut antavat paljon vääriä positiivisia. Testijoukko C on oletettavasti monimutkaisempi ja vääriä positiivisia voidaan olettaa tulevan erittäin vähän.

Kaikki vertailussa mukana olevat tarkastustyökalut antavat listalle käytännössä aina jonkin verran tuloksia, koska työkalut on suunniteltu esittämään eniten toisiaan muistuttavat lähdekoodit ensimmäisenä. Listalle joutumista voidaan yleensä rajoittaa jollain tekaistulla rajalla, mutta listoille päätyy kuitenkin jonkinlaisia tuloksia, vaikka kaikki vertailtavat tiedostot olisivat täysin erilaisia. Lisäksi listaa tulkitsee aina todellinen henkilö, joka tekee työkalujen antamasta raporttilistauksesta päätökset, mitkä parit otetaan mukaan tarkempaan vertailuun. Näistä syistä on vaikeaa esittää kiinteitä rajoja sille, mikä tulkitaan plagiaattihälytykseksi.

Tulosjoukkoja arvioidaan tunnettujen plagiaattien avulla siten, että jos joukoissa A, B ja C työkalu sijoittaa plagiaatin 50 eniten toisiaan muistuttavan tiedostoparin joukkoon, tulkitaan se havaituksi. Nämä perusteet ovat siinä mielessä epärealistisia, että tarkastajan tulisi aina tarkistaa muutama pari myös sellaisten palautusten joukosta, joiden ei enää usko olevan plagiaatteja, jolloin voidaan vahvistaa hyväksyttävyyden uskoa.

Vääriä positiivisia arvioidaan siten, että jos tarkistustyökalu antaa joukkojen A, B ja C vertailussa 50 eniten toisiaan muistuttavan tiedostoparin joukkoon sellaisen parin, jossa toinen osapuolista on kurssihenkilökunnalla teetetty plagiaatti, tulkitaan se vääräksi positiiviseksi. Vääriä positiivisia on varmasti enemmän, mutta koska todellista plagiaattien määrää ei voida tietää, ei voida tietää myöskään todellista väärien positiivisten määrää. Lisäksi arvioidaan oikeiden positiivisten löytämisen helppoutta antamalla arvo kohinalle, joka lasketaan tuloslistalla viimeisen todellisen plagiaattilöydöksen ja listan yläpään väliin jäävät sellaiset parit, joista ei tiedetä, ovatko ne plagiaatteja. Arvoon ei lasketa siis tunnettuja plagiaattipareja eikä sellaisia pareja, jotka ovat plagiaatteja samasta alkuperäisestä ohjelmasta. Kohinan maksimiarvo on tämän vertailun tapauksessa 50, jos yksikään todellinen plagiaattipari ei päädy listalle.

Kohinan laskeminen on suoraviivaista Plakin 2.1 ja Mossin tapauksessa, koska niis-

tä kumpikin antaa tulokset järjestettynä listana. JPlagin tapauksessa tulkinta ei ole niin yksiselitteinen, koska JPlag esittää tulokset kaksiulotteisena mallina yksiulotteisen listan sijaan. Tässä vertailussa JPlagin antamia tulokset järjestettiin samankaltaisuusarvon mukaan ja kaksiulotteisen esitystavan vaikutus käyttäjän kokemaan kohinaan on jätetty huomiotta. Myös Plakin 2.1 ja Mossin tuloslistat riippuvat useammasta kuin yhdestä parametrista, jolloin kohinaan vaikuttaa myös muista tuloksista poikkeavat arvot pelkän järjestyksen mukaan. Kohina antaa kuitenkin jonkinlaisen arvion siitä, kuinka suuresta osasta tuloslistaa voi olettaa löytävänsä plagiaatteja ja kuinka luotettava listan järjestys on. Oletettavasti pienissä ja yksinkertaisissa ohjelmissa kohinaa on huomattavasti, mutta se laskee hyvin nopeasti ohjelmien monimutkaistuessa.

Testijoukkojen A ja B teetetyistä plagiaateista puolet ovat sellaisia, joiden tuottamiseen henkilökunnan edustajat ovat käyttäneet keinoja, joita voidaan olettaa ensimmäisiä ohjelmointikursseja suorittavilta opiskelijoilta, ja toinen puoli sellaisia, joissa henkilökunta on saanut käyttää itse keksimiään naamiointikeinoja parhaansa mukaan. Oletus on, että ensimmäisen puolikkaan plagiaatit ovat tarkastustyökaluille *helppoja* löytää ja jälkimmäisen *vaikeita*.

Testijoukon D vertailun tuloksia ei tulkita samoilla perusteilla kuin joukkojen A, B ja C, vaan joukon D tarkoitus on mitata, voidaanko Plakin 2.1 kaltaista ohjelmistoa käyttää apuna huomattavan suurten tietomäärien seulonnassa ja kuinka paljon yhteiseltä vaikuttavaa ohjelmakoodia joukon D järjestelmät sisältävät. Testijoukon D lähdemateriaali eroaa muista joukoista sekä lähdekoodin kirjoittajien taustojen että ympäristön suhteen. Testijoukon D tapauksessa on ulkopuolisen tarkastelijan mahdotonta tehdä päätelmiä samankaltaisuuksien laillisuudesta, koska kaikkia osapuolten välisiä sopimuksia ei ole julkistettu. Tästä syystä joukon D pareja ei tulkita plagiaateiksi, vaikka niiden samankaltaisuus olisi huomattava. Joukon D osalta tiedostot tulkitaan samankaltaisiksi, jos ne ylittävät vähintään 70% / 70% samankaltaisuuden, eli 70% kummankin tiedoston riveistä pitää löytyä toisesta.

6.4 Tulokset

Tulokset joukolle A ovat nähtävissä taulukossa 6.1. Joukossa A on seitsemän tunnettua plagiaattiparia, joista kaikki vertailutyökalut löysivät samat neljä. Löytämättä jäivät henkilökunnalla teetetyt vaikeat plagioinnit. Huomattavaa tuloksissa on suuret kohina-arvot kaikilla työkaluilla, joka viittaa siihen, että vertailtavat työt muistuttavat huomattavasti toisiaan. Työkalut suoriutuivat hyvin tasaisesti joukon A vertailusta ja erityisen positiivinen huomio on väärien positiivisten puuttuminen kokonaan suuresta kohinasta huolimatta.

Taulukko 6.1: Testijoukko A

Työkalu	Löydetyt	Vääriä positiivisia	Kohina-arvo
Plakki 2.1	4	0	12
Moss	4	0	14
JPlag	4	0	22

Taulukko 6.2: Testijoukko B

Työkalu	Löydetyt	Vääriä positiivisia	Kohina-arvo
Plakki 2.1	4	0	7
Moss	4	1	2
JPlag	3	1	0

Testijoukon B tulokset ovat taulukossa 6.2. Joukossa B on neljä tunnettua plagiaattiparia, joiden etsinnässä työkaluista voidaan havaita eroja. Huomattavaa on Mossin ja JPlagin väärä positiivinen, joka tuli molempien työkalujen tapauksessa samasta parista. JPlag ei löytänyt yhtä vaikeista pareista, Plakki 2.1 ja Moss löysivät kaikki tunnetut plagiaattiparit. Kohinan lasku on joukossa B huomattava. Plakki 2.1:n muita korkeampi kohina-arvo johtuu osittain siitä, että Plakki 2.1 ei painota tiedostojen kokoa tuloslistaa järjestettäessä, ja listalle päätyi muutamia ohjelmien käynnistysrutiineita sisältäviä yksinkertaisia tiedostoja.

Joukon C tulokset ovat taulukossa 6.3. Joukon C tapauksissa työkalujen suoriutumisessa ei ole eroja, vaan kaikki työkalut löysivät joukon ainoan plagiaattiparin ilman kohinaa.

Joukon D vertailu oli alun perin tarkoitus suorittaa Plakin versiolla 1.0, mutta tämä osoittautui niin hitaaksi operaatioksi, että uusi versio ja tämä diplomityö päätettiin tehdä. Plakin versioiden 1.0 ja 2.1 suorituskyvyn vertailua varten joukon D vertailu suoritettiin uudellakin järjestelmällä vain yhteisten rivien algoritmilla, koska versiossa 1.0 pisimmän yhteisen osajonon algoritmia ei vielä ollut toteutettu. Plakin versiolla 1.0 pisimmän yhteisen osajonon vertailu olisi ollut käytännössä mahdotonta, koska yhteisten rivien algoritmilla suoritukseen kului 14 vuorokautta ja 22 tuntia. Uudella versiolla vertailuosaan aikaa kului 28 sekuntia. Vertailuosan lisäksi aikaa kuluu myös tiedostojen muuntamiseen, joka jätettiin pois edellä mainituista ajoista, koska muunnokset olivat uudessa ja vanhassa versiossa samat.

Taulukko 6.3: Testijoukko C

Työkalu	Löydetyt	Vääriä positiivisia	Kohina-arvo
Plakki 2.1	1	0	0
Moss	1	0	0
JPlag	1	0	0

Annetuilla ehdoilla testijoukon D tulosjoukkoon päätyi 230 tiedostoa. Suhteellisesti yhteiseltä vaikuttavaa ohjelmakoodia on siis Androidissa 3,2% ja Java SE:ssä 4,1%. Plakin 2.1 suorittaman vertailun lisäksi osa tulosjoukosta tarkistettiin käsin. Osoittautui, että vertailtavat järjestelmät sisältävät paljon samaa ohjelmakoodia, joista suuri osa on avointa lähdekoodia ja kenen tahansa käytettävissä. Osa samoista tiedostoista sisälsi kuitenkin keskenään erilaisen lisenssimerkinnän, josta on esimerkkinä molemmista järjestelmistä löytyneen tiedoston ArrayBlockingQueue.java lisenssimerkintä Androidin versiona 6.3 ja Java SE:n versiona 6.4

```
/*
* Written by Doug Lea with assistance from members of JCP JSR-166
* Expert Group and released to the public domain, as explained at
* http://creativecommons.org/licenses/publicdomain
*/
```

Kuva 6.3: Lisenssimerkintä Android-järjestelmän tiedostosta ArrayBlockingQueue.java

```
/*
* @(#)ArrayBlockingQueue.java 1.9 04/06/14
*
* Copyright 2004 Sun Microsystems, Inc. All rights reserved.
* SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
*/
```

Kuva 6.4: Lisenssimerkintä Java SE-järjestelmän tiedostosta ArrayBlockingQueue.java

Tiedostojen alkuperään tai omistussuhteeseen ei oteta tässä työssä kantaa, mutta tiedoston ArrayBlockingQueue.java tapauksessa sama tiedosto löytyy myös muualta samalla lisenssillä kuin Android-alustan tiedostossa [7].

7. YHTEENVETO

Tässä diplomityössä käsiteltiin plagioinnin automaattista etsintää ja tunnistusta lähdekooditiedostoista. Työssä esitettiin vaatimukset plagioinnin tunnistamiselle, kuvattiin plagioinnin etsinnän prosessi koulutusorganisaatioiden käyttöön ja toteutettiin plagioinnin etsintään soveltuva työkalu. Työssä arvioitiin työkalun kykyä löytää plagiaatteja TTY:n opiskelijoiden tekemien harjoitustöiden joukosta vertaamalla työkalun tuloksia kahden muun vastaavan työkalun antamiin tuloksiin.

Plagioinnin etsinnän prosessi vaatii suorittajaltaan asiantuntemusta ja järjestelmällisyyttä. Etsinnän suorittajan tulee tuntea plagiointikeinot ja plagiointiin johtavat syyt, jotta suorittajan olisi mahdollista etsiä ja löytää vakuuttavia todisteita. Plagioinnista epäiltäessä pitäisi pyrkiä niin vakuuttaviin todisteisiin, että plagioinnista epäilty tunnustaa plagioinnin, jos on siihen syyllistynyt. Tämä on mahdollista, jos plagioinnin etsinnän prosessi suoritetaan yhtenäisesti ja järjestelmällisesti kuten tässä työssä on esitetty. Plagioinnin etsinnän prosessi koostuu materiaalin keräyksestä, analyysistä, varmistamisesta ja seurausten toimeenpanosta. Materiaalia kerättäessä pyritään ottamaan käsittelyyn kaikki sellainen lähdekoodi, josta plagiointia voi löytyä tai joka voi olla plagioinnin kohteena. Usein lähdemateriaali voidaan rajata tarkasti, esimerkiksi yliopiston ohjelmointikurssien harjoitustöihin.

Plagioinnin tunnistuksen keskeisin ongelma on plagioidun lähdekoodin erottaminen suuresta joukosta aidosti toisistaan riippumattomia lähdekoodeja. Erottaminen on vaikeaa, koska plagioijat yrittävät usein naamioida lähdekoodia siten, että lähdekoodin alkuperä ei olisi enää tunnistettavissa. Eri henkilöiden kirjoittama lähdekoodi voi myös muistuttaa toisiaan sattumalta. Tunnistaminen voidaan tehdä automaattisesti arvioimalla lähdekoodiparien rakenteen samankaltaisuutta erilaisilla lähdekoodia analysoivilla algoritmeilla. Samankaltaisuuden arviointi perustuu nykyaikaisissa järjestelmissä lähdekoodin muuttamiseen yleistettyyn, lähdekoodin rakennetta kuvaavaan muotoon, jolloin lähdekoodin ulkoasulla ei ole merkitystä samankaltaisuuden kannalta.

Lähdekoodiparin korkea samankaltaisuus ei välttämättä tarkoita, että plagiointia olisi todella tapahtunut. Tämän vuoksi automaattisen etsinnän tulokset pitää aina tarkistaa käsin plagioinnin varmistamiseksi. Varmistus tapahtuu etsimällä plagioin-

tiin viittaavia todisteita muuntamattomista lähdekooditiedostoista. Varmistuksen perusteella todetun plagiointiepäilyn osapuolet kutsutaan kuulemistilaisuuteen, jossa osapuolet voivat esittää oman kantansa epäilyyn liittyen. Kuulemistilaisuuden perusteella voidaan tehdä päätös mahdollisista jatkotoimenpiteistä tai todeta, että plagiointia ei ole tapahtunut.

Diplomityön yhtenä tavoitteena oli suunnitella ja toteuttaa plagioinnin automaattiseen etsintään soveltuva vertailutyökalu Plakki 2.1, jossa Plakin 1.0 suorituskykyongelmat olisi korjattu. Plakkia 2.1 käytettiin ensimmäisen kerran vuoden 2011 joulukuussa kurssilla Laaja ohjelmointi 1. Ensimmäiset kokemukset uudesta työkalusta olivat lupaavia vertailun nopeuden ansiosta. Plakki 2.1 paljasti ensimmäisillä ajokerroilla myös vanhoja kopiointitapauksia, joita ei oltu havaittu aiempina vuosina, joten työkalun kyky löytää plagiaatteja on parantunut.

Työssä vertailtiin Plakki 2.1:n kykyä löytää plagiaatteja kahden plagiaatinpaljastimen, Stanfordin yliopistossa kehitetyn Mossin ja Karlsrühen yliopistossa kehitetyn JPlagin suoriutumiseen. Vertailun tavoitteena oli selvittää, kykeneekö Plakki 2.1 löytämään tunnettuja plagiaatteja testijoukoista yhtä hyvin kuin vertailussa mukana olleet muut järjestelmät. Tulosten perusteella vertailussa mukana olleilla järjestelmillä ei ole merkittävää eroa plagiaattien erottelukyvyssä, mutta graafisen lähdekoodien vertailuominaisuutensa vuoksi Moss ja JPlag ovat vertailun varmistusvaiheen osalta tehokkaampia käyttää. Plakkia 2.1 kannattaa kuitenkin kehittää muista hieman eroavan vertailumenetelmän takia, joka ei tämän työn puitteissa tehtyjen testien perusteella antanut yhtäkään väärää positiivista.

Internetin ja sähköisten arkistojen aikakaudella seurauksena tiedon ja lähdemateriaalin saavutettavuus on kasvanut räjähdysmäisesti. Saavutettavuus ja tiedon suuri määrä tekee plagioinnista helppoa. Tämän vuoksi väärinkäytöksiä automaattisesti etsivät järjestelmät ovat tärkeitä ja niitä on syytä kehittää, jotta esimerkiksi yliopistoista valmistuvien taitoihin voidaan luottaa.

KIRJALLISUUTTA

[1] Mike Blake. Reuters: Oracle sues google over android, 2010.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[3] Fintan Culwin and Thomas Lancaster. Plagiarism issues for higher education, 2001.

[4] Ronald I Greenberg. Fast and simple computation of all longest common subsequences, 2002.

[5] The Open Group. Archimate® version 2.0, 2012.

[6] Mike Joy and Michael Luck. Plagiarism in programming assignments. IEEE Transactions on Education, Vol. 42, No. 2, May 1999.

[7] Doug Lea. Jsr-166, javan rinnakkaisuuspakkaus, julkaistu osana jdk5:n pakkausta java.util.concurrent, 2004.

[8] Tutkimuseettinen Neuvottelukunta. Hyvä tieteellinen käytäntö ja sen loukkausepäilyjen käsittely suomessa, 2012.

[9] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. Journal of Universal Computer Science, vol. 8, no. 11 (2002), 2002.

[10] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting, 2003.

[11] Toni Uimonen and Kirsti Alamutka. Ohjelmointiharjoitustoiden kopiopaljastin plakki. In report 27 of TTY Department of Software Systems, 61-64, Tampere, FI, 2001.

[12] Antti Valmari. Ohjelmistotieteen perustyökaluja, laskuharjoitusmoniste, 2012.

[13] G. Whale. Identification of program similarity in large populations. The Computer Journal, VOL33, NO.2, 1990, 1988.

[14] Lofti A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning-i. American Elsevier Publishing Company, Inc., 1975.